Electrical and Computer Engineering Department
University of Maryland
College Park, MD 20742-3285
Glenn L. Martin Institute of Technology ♦ A. James Clark School of Engineering
Dr. Charles B. Silio, Jr.
Telephone 301-405-3668
Fax 301-314-9281
silio@eng.umd.edu

The Microarchitecture/Microprogramming Level

These notes are based on and extend material in Chapter 4 of A. S. Tanenbaum, *Structured Computer Organization*, 3rd Edition, Prentice Hall, 1990. The accumulator based machine whose instruction set architecture is called the Mac-1 has its data path microarchitecture and its microprogrammed implementation (called the Mic-1) presented here. This presentation differs from the stack oriented IJVM and corresponding Mic-1 in Tanenbaum's 5th Edition textbook.

One of the differences between the Mic-1 (microprogrammed computer) presented here and the one in the 5th Edition textbook is that all registers in this Mic-1 are constructed from clocked (or gated) D-latches, as shown in Fig. 1; whereas, registers in the 5th Edition text use edge-triggered flip-flops. Fig. 2 shows how an 8-bit register is built using clocked D-latches and three-state (i.e., tri-state) buffers for connection to two output buses.

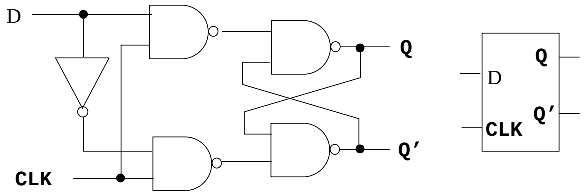


Figure 1: Clocked D-latch

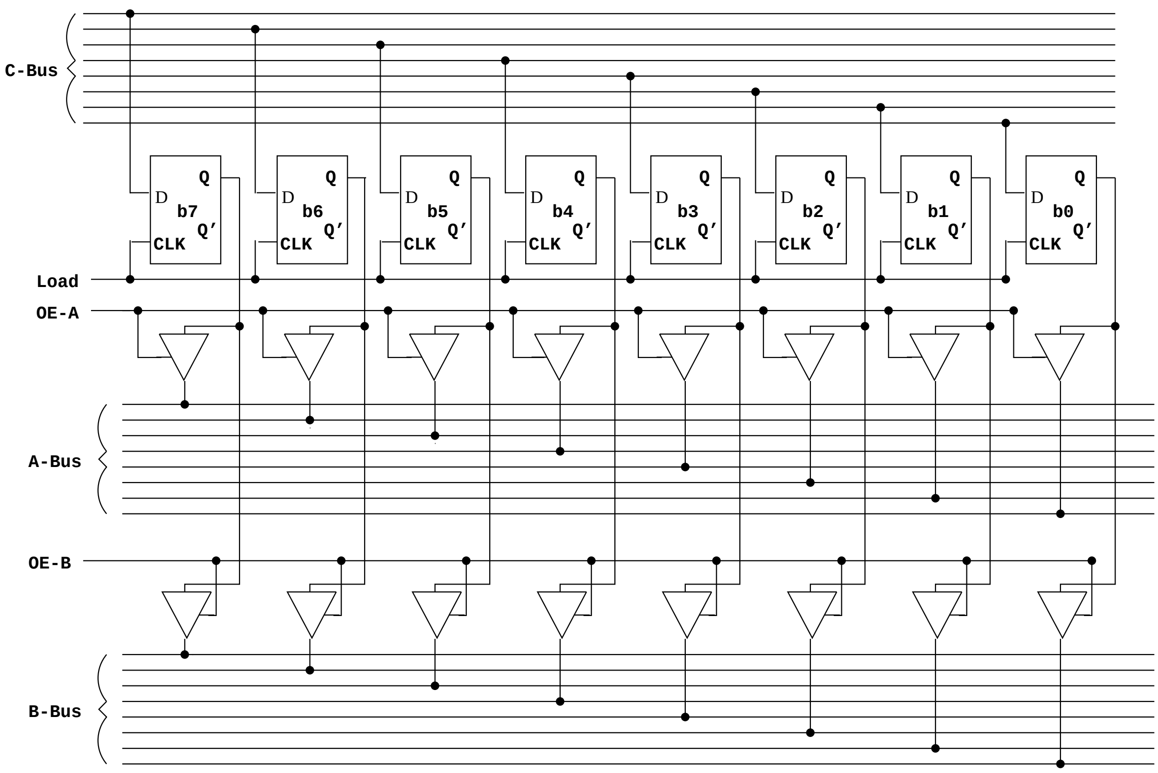


Figure 2: Eight-bit register and bus connections

Registers: A register is a device capable of storing information. Conceptually, registers are the same as main memory, the difference being that the registers are located physically within the processor itself, so they can be read from and stored into faster than words in main memory, which is usually off-chip. Larger and more expensive machines usually have more registers than smaller and cheaper ones, which must use main memory for storing intermediate results. On some computers a set of registers numbered 0, 1, 2, ..., $n-1$, is available at the microprogramming level and is called **local storage** or **scratchpad storage**.

A register can be characterized by a single number: namely, how many bits can it hold (e.g., Fig. 2 is an 8-bit register). The bits (binary digits) in an n-bit register could be numbered from left to right or from right to left. The numbering convention assumed in these notes for the bits in an n-bit register is right to left from 0 to $n-1$ in the natural powers of two order of a positional number system for integers. In other words, bit 0 is stored in the rightmost D-latch in Fig. 2 and bit 7 is stored in the leftmost D-latch (which corresponds to bit $n-1$ when $n=8$).

Information placed in a register remains there until some other information replaces it. The process of reading information out of a register does not affect the contents of the register. In other words, when a register is read, a copy is made of its contents and the original is left undisturbed in the register. Similarly, when information is moved from one register to another, a copy is loaded into the destination register and the contents of the source register remain undisturbed.

Buses: A bus is a collection of wires used to transmit signals in parallel. For example, buses are used to allow the contents of one register to be copied to another one. A bus may be unidirectional or bidirectional. A unidirectional bus can transfer data only in one direction; whereas, a bidirectional bus can transfer data in either direction but not both simultaneously. Unidirectional buses are typically used to connect two registers, one of which is always the source and the other of which is always the destination. Bidirectional buses are typically used when any of a collection of registers can be the source and any other one can be the destination.

Many devices have the ability to connect and disconnect themselves electrically from the buses to which they are physically attached. These connections can be made or broken in nanoseconds. A bus whose devices have this property is called a **tri-state** (or three-state) bus (the term tri-state being a registered trademark of National Semiconductor Corp.). A tri-state buffer amplifier is used to make the connections. These tri-state buffer amplifiers are shown in Fig. 2 as triangular shapes whose inputs come from the output of the D-latch to which each is connected and each of whose outputs is connected to a single bus wire. The other input to the buffer amplifier (labeled either OE-A or OE-B) is a control (or enable) input. If this control input is in the logic zero state, then the output of its buffer amplifier is in the high-impedance state (i.e., disconnected from the bus wire to which it is attached). If the control input is in the logic one state (also called active-high) then the buffer amplifier's output value equals its input value (either logic 0 or logic 1), and the D-latch's output state is connected to the corresponding bus wire.

In most microarchitectures, some registers are connected to one or more input buses and to one or more output buses. Fig. 2 depicts an 8-bit register connected to one input bus and to two output buses. The register has three control inputs: namely, Load, OE-A, and OE-B, where OE stands for "output enable." When "Load" is in the logic zero state, the contents of the register are not affected by the signals on the C-bus wires. When "Load" is raised to the logic 1 state the values on the C-bus wires are copied into their corresponding D-latches in parallel. After the new values are latched "Load" can be returned to its logic zero state, and the register remembers the binary value last loaded into it.

When "OE-A" is at the logic zero level, the register is disconnected from the A-Bus (and similarly for "OE-B" with respect to the B-Bus). When "OE-A" is raised to the logic 1 level, the register is connected to the A-Bus wires (and similarly for "OE-B" with respect to the B-bus).

In order to transfer data from this register to another register R using the A and C buses. The input to register R must be connected to the C-Bus, and "OE-A" for this register must be raised to the logic 1 level in order to place the register's contents on the A-Bus. Other circuitry such as an arithmetic and logic unit (ALU) which is not shown here must then be used to connect the A-Bus wires to the C-Bus wires. After a short time to allow the signals on the buses to settle down and become stable then the Load signal connected to register R is raised to the logic 1 level and the information transfer is accomplished.

Because drawing all of the wires and latches shown in Fig. 2 requires too much space, a shorthand schematic such as that shown in Fig. 3 is used instead. Fig. 3 depicts a 16-bit register that would be constructed internally in the same fashion as the 8-bit register shown in Fig. 2 but with 8 more latches, 16 more buffer amplifiers, and 24 more bus wires.

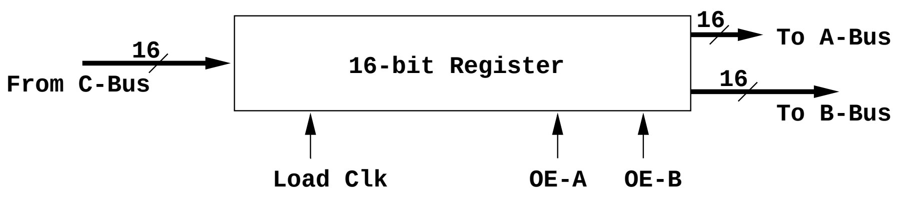


Figure 3: Sixteen-bit register schematic

Decoders and Multiplexers: Circuits that have one or more input lines and compute one or more output values that are uniquely determined by the present inputs are called **combinational circuits**. Two important combinational circuits are decoders and multiplexers. A **decoder** has n input lines and 2^n output lines numbered 0 to $2^n - 1$. If the binary number on the input lines has decimal value k, then output line number k takes the value 1 and all other output lines take the value 0. A decoder always has exactly one output line whose value is set to 1, with all the rest set to 0. A **multiplexer** has 2^n data inputs (either individual lines or buses), one data output of the same width as the inputs, and an n-bit control input that is internally decoded to select one of the inputs and route it to the output. The structure of a 2 to 1 multiplexer is shown in Fig. 4. If instead of a single input line one wishes to switch the contents of one of two n-bit input buses to an n-bit output bus, then one must use n 2 to 1 multiplexers (one per output bit line) all selected by the same selection input value S.

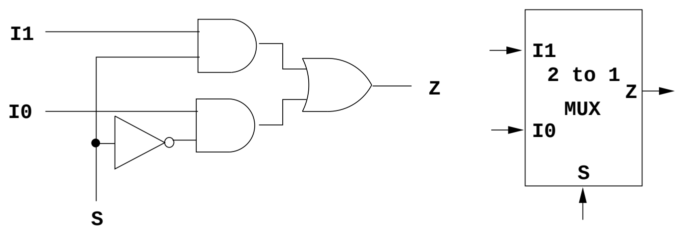


Figure 4: 2 to 1 Multiplexer (one for each output bit when used with registers)

An Example Microarchitecture

The **data path** of our example microarchitecture is shown in Fig. 5. The data path is that part of the central processing unit (CPU) that contains the arithmetic and logic unit (ALU) and its inputs and outputs. In this case it contains 16 identical 16-bit registers, labeled PC, AC, SP, and so on, that form a scratchpad memory accessible only to the the microprogramming level. The registers labeled 0, +1, and -1 will be used to hold the indicated constants (with -1 in two's complement form). The meaning of the other register names will be explained later. Each register can output its contents onto one or both of two internal buses, the A-Bus and the B-Bus, and each can be loaded from a third internal bus, the C-Bus as shown in the figure.

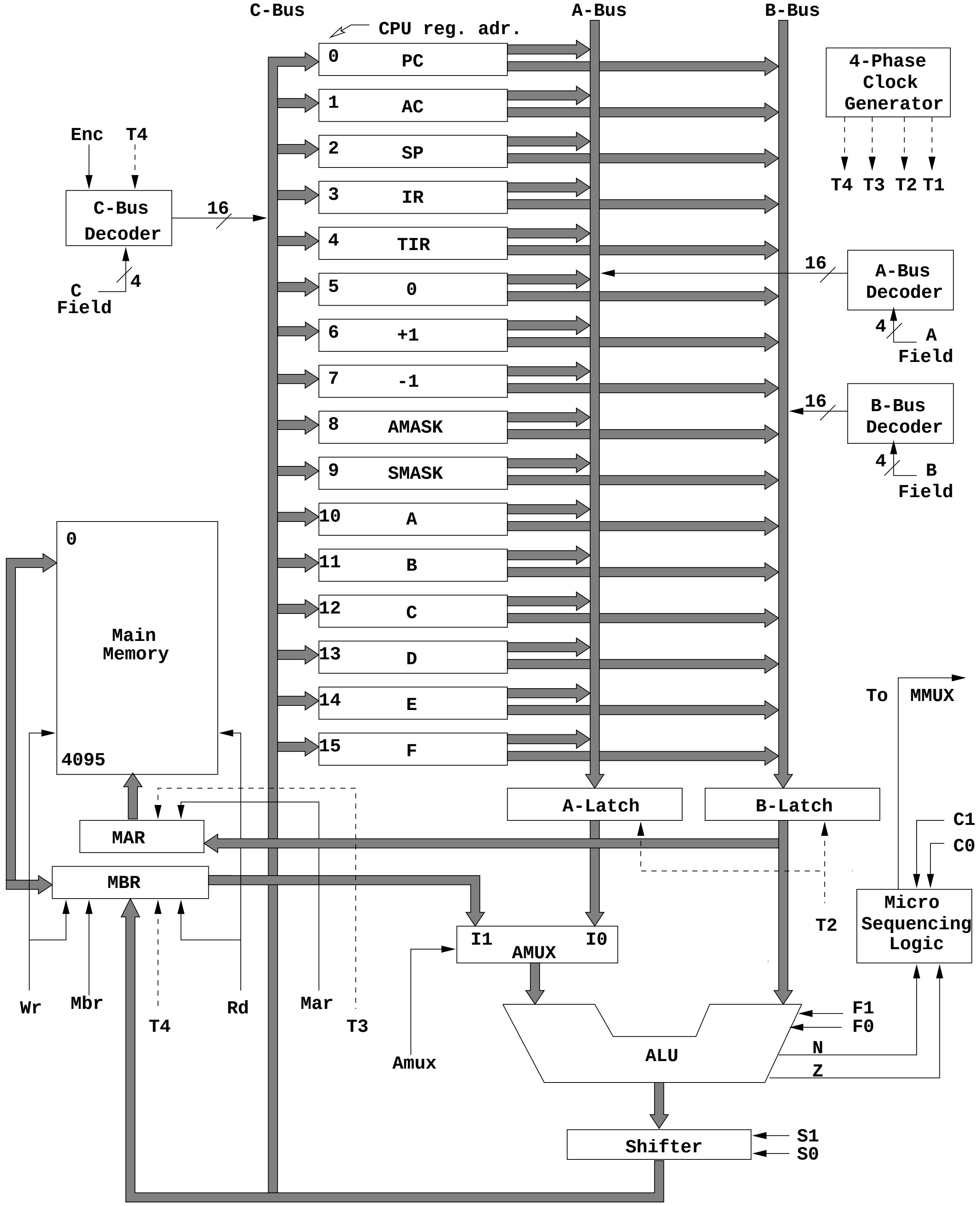


Figure 5: The data path for example microarchitecture (Mic1/Mac1)

The A and B buses respectively feed the left and right inputs of a 16-bit wide ALU that can perform four functions: addition (A + B), bitwise logical AND (A.AND.B), left input straight-through (A), and bitwise logical complement (i.e., 1's complement) of the content of the left input (NOT A). The function to be performed is specified by the two ALU control lines F_1 and F_0. The ALU generates two status bits based on the current ALU output: N, which takes the value 1 when the ALU output is negative, and Z, which takes the value one when the ALU output is zero. The N bit is just a copy of the high-order (bit position 15) output bit. The Z bit is the NOR of all the ALU output bits (namely, bits 0 through 15).

The 16-bit ALU output goes into a shifter, which is a combinational circuit that can logically shift its input 1 bit left or right, or not at all, and gate the result to its 16-bit output. The function to be performed by the shifter is specified the the two shifter control lines S_1 and S_0. It is possible to perform a 2-bit left shift of a register, R, by using the ALU to compute R + R (which is a 1-bit left shift) and then shifting this sum another bit left using the shifter.

The A-Bus decoder is used to decode a 4-bit register designator (A-field) that selects one of the 16 scratchpad registers to be gated onto the A-Bus. The outputs of the decoder are 16 output enable (OE-A) signals (one for each register) and one and only one of the OE-A signals takes the value 1. The B-Bus decoder is used to decode a 4-bit register designator (B-field) that selects one of the 16 scratchpad registers to be gated onto the B-Bus. The outputs of the decoder are 16 output enable (OE-B) signals (one for each register) and one and only one of the OE-B signals takes the value 1. The C-Bus decoder is used to decode a 4-bit C-field register designator that selects the scratchpad register to be loaded from the C-Bus. The outputs of the C-Bus decoder are 16 load_clock signals (one for each register). Because all 16 possible C-field values are assigned to the 16 registers, an additional control input is needed to prevent loading any of the registers. This additional control input is ENC (for enable-C). If ENC = 0, then all 16 of the decoder's load outputs remain at logic level zero, and none of the registers is overwritten. If ENC = 1, then one and only one of the destination registers sees a load_clock line = 1 at the appropriate time determined by yet another control input called T4.

Neither the A-Bus nor the B-Bus feeds the ALU directly. Instead, each one feeds a latch (i.e., a register) that in turn feeds the ALU. The latches are needed because the ALU is a combinational circuit – it continuously computes the output for the current input and function code. Feeding the left and right ALU inputs directly from the A and B buses (without the additional latches) can cause race problems. For example, consider assigning to the destination register A the sum of the contents of registers A and B, denoted A:= A + B. As A is being written into, the value on the A-Bus begins to change, which causes the ALU output and thus the contents of the C-Bus to change as well. Consequently, the wrong value may be stored into A. In other words, in the assignment A:= A + B, the old A on the right-hand side is the original A value, not some bit-by-bit mixture of the old and new values. By inserting latches (namely, the A-latch and B-latch) into the A and B buses, we can freeze the original A and B values there early in the cycle, so that the ALU is shielded from changes on the buses as the new value is being stored into the scratchpad.

One can think of the A-latch and the B-latch as shared slave latches for the correspondingly selected source master latches in the scratchpad. This saves using slave latches in each scratchpad register that using master-slave flip-flops to build the registers would require, but it complicates the timing somewhat. The A-latch and B-latch are loaded by timing control signal T2 that is generated by a 4-phase clock generator circuit shown in Fig. 6.

Computer circuits are normally driven by a **clock**, a device that emits a periodic sequence of pulses. These pulses define machine cycles. During each machine cycle, some activity occurs, such as the execution of a microinstruction. It is often useful to divide a cycle into subcycles so different parts of the microinstruction can be performed in a well-defined order. For example, the inputs to the ALU must be made available and allowed to become stable before the output can be stored.

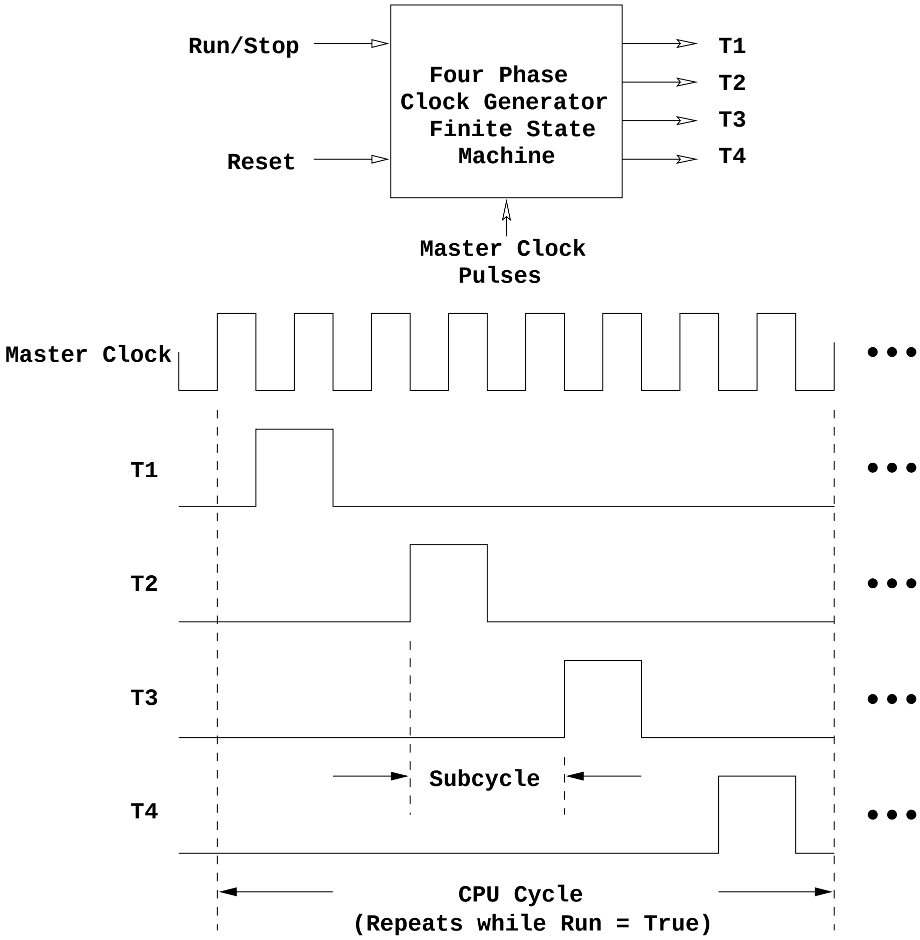


Figure 6: Four Phase Clock Cycle

Main Memory: Processors need to be able to read data from memory and write data to memory. Most computers have an address bus, a data bus, and a control bus for communication between the CPU and memory. To read from memory, the CPU puts a memory address on the address bus and sets the control signals appropriately, for example by asserting "Rd" (READ). The memory then puts the requested item on the data bus. In some computers memory read/write is synchronous; that is, the memory must respond within a fixed time. This is what we assume for our microarchitecture; namely, the memory must respond within four clock (subcycle) ticks. On other computers, the memory may take as long as it wants, signaling the presence of data using a (e.g., READY or memory function complete) control line when it is finished.

Writes to memory are done similarly. The CPU puts the data to be written on the data bus and the address to be stored into on the address bus and then it asserts "Wr" (WRITE). (An alternative to having "Rd" and "Wr" is to have MREQ, which indicates that a memory request is desired, and R/W, which distinguishes read from write. In either case two control lines are required.)

On most machines (except for our example) a memory access is nearly always considerably longer than the time required to execute a single microinstruction. Consequently the microprogram must keep the correct values on the address and data buses for several microinstructions (i.e., machine cycles). To simplify this task, it is often convenient to have two registers, the **MAR** (Memory Address Register) and the **MBR** (Memory Buffer Register), that drive the address and data buses, respectively. Both registers sit between the CPU and the system's memory bus. The address bus is unidirectional on both sides and is loaded from the CPU side when the "Mar" control line is asserted. The output to the system address lines is always enabled (as is the case here) [or possibly only during reads and writes, which requires an output enable line driven by the OR of "Rd" and "Wr" (not shown)]. Because the main memory in our example microarchitecture has only 4096 16-bit words, the MAR is a 12-bit register. In our example microarchitecture the MAR is connected to the B-Bus (rather than to the C-Bus) so that both the MAR and the MBR can be loaded in the same machine cycle (i.e., loaded by the same microinstruction) Because the MAR is connected to the B-Bus (rather than to the C-Bus) and because of the way the microprogram controlling this machine is written, it is restricted in size to 12-bits. To allow for a 16-bit MAR connected in this way would require two machine cycles (i.e., two microinstructions) and the use of another scratchpad register to properly load all 16 bits into the MAR. This is a consequence of the design choices made by others; namely, the author of the text from which this example is derived. A more detailed schematic of the MAR and its connections is shown if Fig. 7.

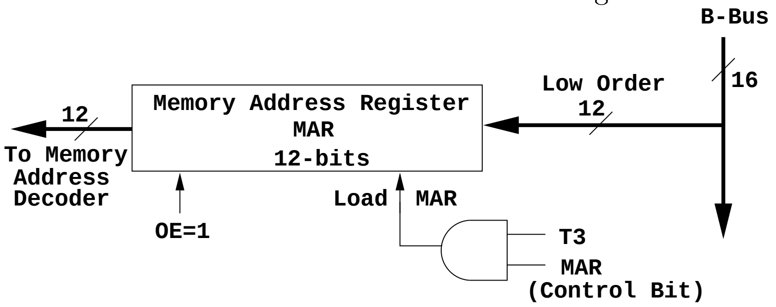


Figure 7: Memory Address Register (MAR)

As shown in Fig. 8, the "Mbr" control line causes the MBR to be loaded from the C-Bus on the CPU side. The MBR output is always enabled on the CPU side and is presented to a 2 to 1 multiplexer (the AMUX) that switches the input to the left ALU input between the A-latch and the MBR under control of a signal called Amux. If Amux = 0, the left input of the ALU sees the contents of the A-latch and if Amux = 1, it sees the contents of the MBR. The system's memory data bus is bidirectional, and the "Rd" and "Wr" control signals are used to determine its direction between memory and the MBR (to memory on write and from memory on read).

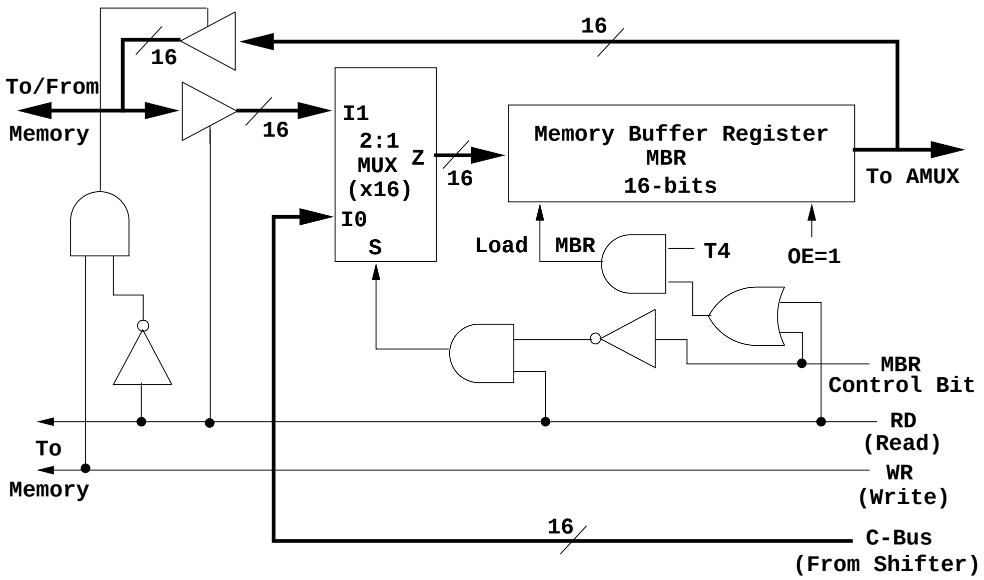


Figure 8: Memory Buffer Register (MBR)

To control the data path of our microarchitecture in Fig. 5 requires 60 signals that belong to the following nine functional groupings:

16 signals to control loading the A-Bus from the scratchpad

16 signals to control loading the B-Bus from the scratchpad

16 signals to control loading the scratchpad from the C-Bus

1 signal to control loading the A and B latches

2 signals to control the ALU function

2 signals to control the shifter

4 signals to control the MAR and MBR

2 signals to indicate memory read or memory write

1 signal to control the AMUX

Given the values of the 60 signals, we can perform one cycle of the data path. A cycle consists of gating values onto the A and B buses, latching them in the two bus latches, running the values through the ALU and shifter, and finally storing the results in the scratchpad and/or the MBR. In addition, the MAR can also be loaded, and a memory cycle initiated. As a first approximation we could have a 60-bit control register, with one bit for each control signal. A 1 bit means that the signal is asserted and a 0 means that it is not asserted (i.e., negated). We also need a multiphase clock generator circuit to control when things happen during the cycle.

However, at the price of a small increase in circuitry, we can greatly reduce the number of bits needed to control the data path. Using all 16-bits to control the A-Bus would allow 2^{16} combinations of signal values, only 16 of which are valid because the scratchpad has only 16 registers. Therefore, we can encode the A-Bus control information in a 4-bit field and use a decoder to generate the 16 control signals. The same holds true for the B-Bus.

The situation is slightly different for the C-Bus. In principle, multiple simultaneous stores into the scratchpad are feasible, but in practice this feature is only infrequently useful, and most hardware designs do not provide for it. Therefore, we will also encode the C-Bus control into a 4 bit field. Having encoded some of the control signals into fields and in turn supplied corresponding decoder circuits, we have saved 3×12 bits. We now need only 24 bits to control the data path. Because the A and B latches are always loaded at a certain point in time, we can supply a multiphase clock generator circuit and use one of the clock phases (say T2) as this control input, leaving 23 control bits needed. After the values in the A and B latches settle down the MAR can be clocked (at subcycle time T3) to copy the content of the B latch if the "Mar" control bit is set to 1. More time is needed, however, for the data signals to propagate through the ALU and shifter circuitry before they have settled down and can be copied from the C-Bus into their destination(s) in the scratchpad or MBR at subcycle time T4. One additional signal that is not strictly required, but is often useful, is one to enable/disable storing the C-Bus into the scratchpad. In some situations one merely wishes to perform an ALU operation to generate the N and Z signals, but does not wish to store the result. With this extra bit, which we will call ENC (ENable C), we can indicate that the C-Bus contents are to be stored (ENC $= 1$) or not (ENC $= 0$).

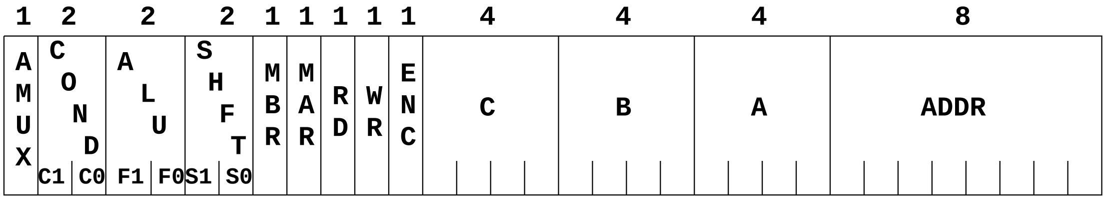
With ENC included we can control the data path with a 24 bit number. Now we note that "Rd" and "Wr" can be used to control the latching of the MBR from the system's memory data bus and the enabling of the MBR onto it, respectively (as shown in Fig. 8). This observation reduces the number of independent control signals needed from 24 down to 22.

The next step in the design of the microarchitecture is to invent a microinstruction format containing 22 bits. Fig. 9 shows such a format with two additional fields COND and ADDR, which will be described shortly. The microinstruction contains 13 fields, 11 of which are as follows:

AMUX	–	controls left ALU input: 0 = A-latch, 1 = MBR
ALU	–	ALU function: 0 = A + B, 1 = A.AND.B, 2 = A, 3 = $\overline{\text{A}}$
SHFT	–	shifter function: 0 = no shift, 1 = right shift, 2 = left shift
MBR	–	loads MBR from shifter: 0 = don't load MBR, 1 = load MBR
MAR	–	loads MAR from B-latch: 0 = don't load MAR, 1 = load MAR
RD	–	requests memory read: 0 = no read, 1 = load MBR from memory
WR	–	requests memory write; 0 = no write, 1 = write MBR to memory
ENC	–	controls storing into scratchpad: 0 = don't store, 1 = store
C	–	selects register for storing into if ENC = 1: 0 = PC, 1 = AC, etc.
B	–	selects B-Bus source: 0 = PC, 1 = AC, 2 = SP, 3 = IR, etc.
A	–	selects A-Bus source: 0 = PC, 1 = AC, 2 = SP, 3 = IR, etc.

Microinstruction Format (32-bit word)

Number of bits in each field:



AMUX
0 = A-latch
1 = MBR

COND
C1 C0
0 0 = No Jump
0 1 = Jump if N=1
1 0 = Jump if Z=1
1 1 = Jump always

ALU
F1 F0
0 0 = A + B
0 1 = A and B
1 0 = A
1 1 = $\overline{\text{A}}$

SHFT
S1 S0
0 0 = No shift
0 1 = Shift right 1 bit
1 0 = Shift left 1 bit
1 1 = (not used)

MBR, MAR, RD, WR, ENC
0 = No
1 = Yes

Figure 9: Microinstruction Format (32-bits) for Mic-1/Mac-1 microarchitecture

The ordering of the fields is completely arbitrary. This ordering has been chosen to minimize line crossings in a subsequent figure. (Actually, this criterion is not as crazy as it sounds; line crossings in figures usually correspond to wire crossings on printed circuit boards or on integrated circuit chips, which cause difficulties in two-dimensional designs.)

Microinstruction Timing: Although our discussion of how a microinstruction can control the data path during one cycle is almost complete, we have mostly neglected one issue up until now: timing. A basic ALU cycle consists of setting up the A and B latches, giving the ALU and shifter time to do their work, and storing the results. It is obvious that these events must happen in that sequence. If we try to store the C-Bus contents into the scratchpad before the A and B latches have been loaded, garbage will be stored instead of useful data. To achieve the correct event sequencing, we use a four-phase clock, that is a clock with four subcycles, as shown in Fig. 6. The key events during each of the four subcycles are as follows:

1. Load the next microinstruction to be executed into a register called **MIR**, the MicroInstruction Register.

2. Gate selected scratchpad registers onto the A and B buses and capture them in the A and B latches.

3. Now that the inputs are stable, give the ALU and shifter time to produce a stable output and load the MAR if required.

4. Now that the shifter output is stable, store the C-Bus contents into the scratchpad and load the MBR, if either is required.

Fig. 10 presents a detailed block diagram of the complete microarchitecture of our example machine. It may look imposing initially, but it is worth studying carefully. When you fully understand every box and every line on it, you will be well on your way to understanding the microprogramming level. The block diagram has two parts, the data path on the left, which we have already discussed in detail, and the control section on the right, which we will now examine.

The largest and most important item in the control portion of the machine is the **control store**. This special, high-speed memory is where the microinstructions are kept. On some machines it is read-only memory (ROM); on others it is read/write memory. In our example, microinstructions are 32 bits wide and the microinstruction address space consists of 256 words, so the control store occupies a maximum of $256 \times 32 = 8192$ bits. By comparison, the Digital Equipment Corporation (DEC) PDP-11/40 was a popular and commercially successful microprogrammed minicomputer in the mid 1970's that also had a 256 word control store, but its microinstructions were 56 bits wide.

Like any other memory, the control store needs an MAR and an MBR. In this case we will call the MAR the **MPC** (MicroProgram Counter) because its only function is to point to the next microinstruction to be fetched from the memory for execution. The MBR is just the **MIR** as mentioned above. In this microarchitecture the control store and the main memory are different entities; the control store holds the microprogram and the main memory holds the conventional machine language program.

From Fig. 10 it is clear that the control store continuously tries to copy the microinstruction addressed by the MPC into the MIR. However, the MIR is loaded only during subcycle 1, as indicated by the dashed line from clock output T1 to it. During the other three subcycles of the clock, it is not affected, no matter what happens to the MPC.

During subcycle 2 (which lasts between the rising edge of T1 and the rising edge of T2) the MIR becomes stable, and the various fields begin controlling the data path. In particular the A and B fields select the scratchpad registers to be gated onto the A and B buses, respectively. The A and B decoder boxes provide for the 4-to-16 decoding of each field needed to drive the OE-A and OE-B lines at the scratchpad registers (see Fig. 3). Clock signal T2 loads the A and B latches, which after their outputs settle, provide stable ALU inputs for all remaining subcycles during the rest of the cycle. While data are being gated onto the A and B buses, the increment unit in the control section of the machine computes MPC + 1, in preparation for loading the next sequential microinstruction during the next cycle. By overlapping these two oprations, instruction execution can be speeded up.

In subcycle 3, the ALU and shifter are given time to produce valid results. The AMUX microinstruction field determines the left input to the ALU; the right input always comes from the B-latch. Although the ALU is a combinational circuit, the time it takes to compute the sum is determined by the carry-propagation time, not the normal gate delay. The carry-propagation time is proportional to the number of bits in the word. While the ALU and shifter are computing, the MAR is loaded from the output of the B-latch at T3 if the MAR field in the microinstruction is 1.

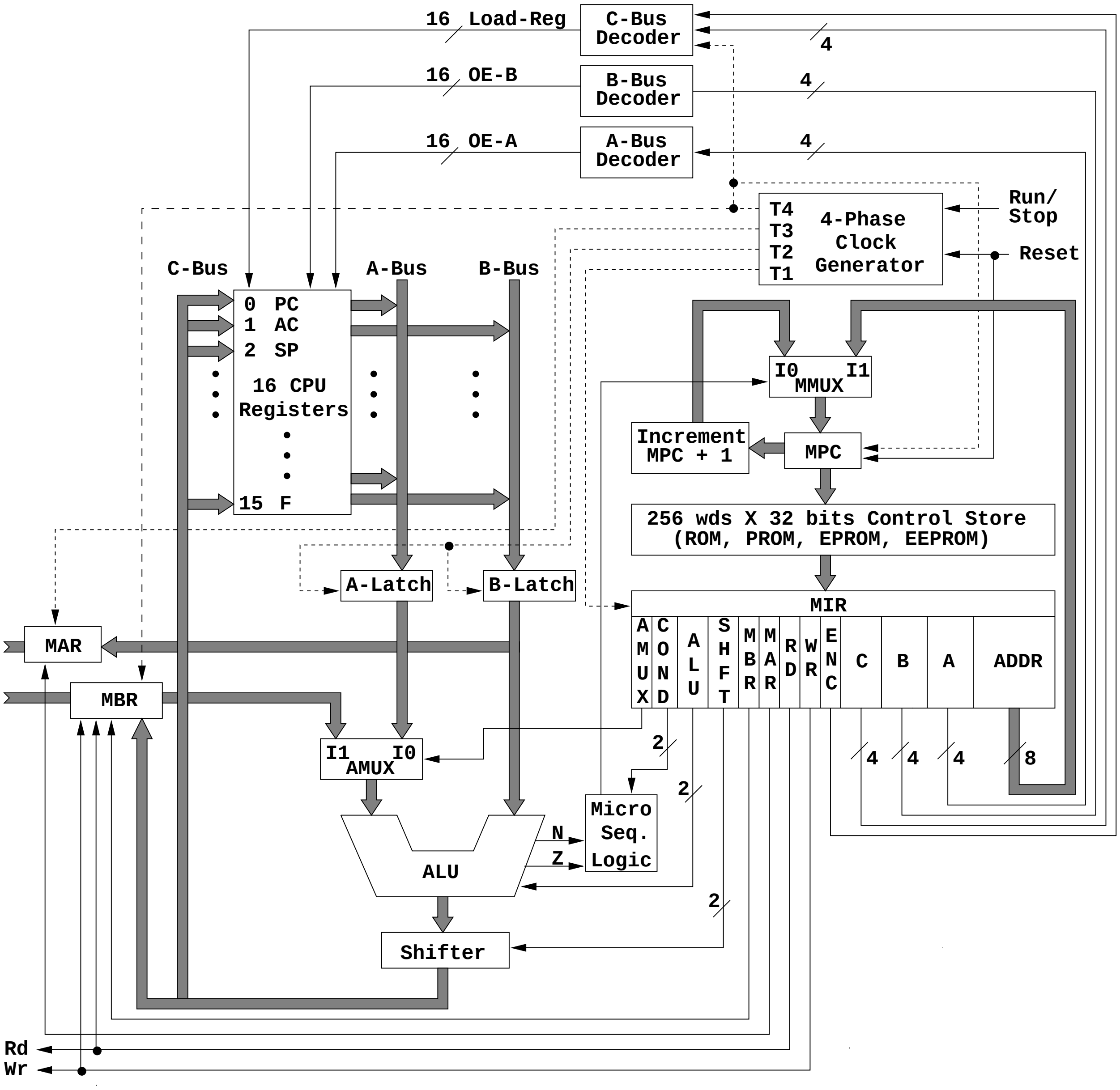


Figure 10: The complete block diagram for example microarchitecture (Mic-1/Mac-1)

During the fourth and final subcycle, the C-Bus may be stored back into the scratchpad and MBR, depending on ENC and MBR. The box labeled "C decoder" takes ENC, T4, and the C field from the microinstruction as inputs and generates the one (or none) of the 16 register load signals. Internally it performs a 4-to-16 decode of the C field and then ANDs each of these 16 signals with a signal derived from ANDing subcycle 4 line T4 with ENC. Thus, a scratchpad register is loaded only if three conditions prevail:

1. ENC = 1.

2. It is subcycle 4 with T4 = 1.

3. The register has been selected by the C field.

The MBR is also loaded during subcycle 4 if MBR = 1.

Microinstruction Sequencing: The only remaining issue is how the next microinstruction is chosen. Although some of the time it is sufficient just to fetch the next microinstruction in sequence, some mechanism is needed to allow conditional jumps in the microprogram in order to enable it to make decisions. For this reason two fields are provided in each microinstruction; namely, ADDR, which is the 8-bit address of a potential successor to the current microinstruction, and COND, which determines whether the next microinstruction is fetched from the control store address that is one greater than the contents of the current MPC (i.e., MPC + 1) or from the location specified by the ADDR field. Every microinstruction potentially contains a conditional jump. The decision to allow for this in the microinstruction format was made because conditional jumps are very common in microprograms, and allowing every microinstruction to have two possible successors makes them run faster than the alternative of setting up some condition in one microinstruction and then testing it in the next.

The choice of address from which the next microinstruction will be fetched is determined by the box labeled "Micro Sequencing Logic" during subcycle 4, when the ALU output signals N and Z are valid. The output of this box controls the M multiplexer (MMUX), which routes either MPC + 1 or ADDR to the MPC (loaded by clock signal T4) where it will direct the fetching of the next microinstruction. The desired choice is indicated by the setting of the COND field as follows:

0 = Do not jump: next microinstruction is taken from MPC + 1

1 = Jump to ADDR if N = 1

2 = Jump to ADDR if Z = 1

3 = Jump to ADDR unconditionally

The Micro Sequencing Logic combines the two ALU bits, N and Z, and the two COND bits C_1 and C_0 to generate an output that is then used as the selection input to the MMUX. The Boolean expression for generating the selection signal (Mmux) is:

$$\mathrm{Mmux} = \overline{C_1} C_0 N \vee C_1 \overline{C_0} Z \vee C_1 C_0 = C_0 N \vee C_1 Z \vee C_1 C_0$$

where "$\vee$" means logical OR. In words, the selection control signal to the MMUX is 1 (routing ADDR to MPC) if C_1C_0 is 01_2 and N = 1, or C_1C_0 is 10_2 and Z = 1, or C_1C_0 is 11_2. Otherwise, it is 0 and the next microinstruction in sequence is fetched.

Because the MAR is loaded at time T3, the memory control unit will not have enough time to decode the address specified and either read from or write to it when clock pulse T4 comes along. In fact, during a memory read the MBR will be loaded with garbage by the first T4 clock pulse following the loading of the MAR at T3. Hence, if a microinstruction starts a main memory read, by setting "Rd" to 1, it must also have Rd = 1 in the next microinstruction executed (which may or may not be located at the next control store address). In other words, "Rd" must be set to 1 in two consecutive microinstructions in order for the MBR to be loaded with correct data (returning from main memory) by the second T4 clock pulse following the loading of the MAR at T3. A full four clock ticks (corresponding to a full microinstruction cycle time) are needed for the main memory to respond with valid data. Thus, the data become available two microinstructions after the read was initiated. If the microprogram has nothing else useful to do in the microinstruction following the one that initiated a memory read (or write), that microinstruction's only task is then to keep Rd = 1 (or for writes Wr = 1). In the same way, a memory write also takes two microinstruction times to complete. In the microinstruction initiating the write the MAR is typically loaded with the address into which data will be written at clock pulse T3, and the data to be written are loaded into the MBR at clock pulse T4. The main memory again needs four clock ticks to decode the address and complete the write. Thus, "Wr" must be set equal to 1 in two consecutive microinstructions (the one initiating the write and the one following it in time).

An Example Macroarchitecture, the Mac-1

We now consider the instruction set architecture of the conventional machine level to be supported by the microprogrammed interpreter running on the machine of Fig. 10. For convenience, we will call the architecture of the level 2 or 3 machine the **macroarchitecture** to contrast it with level 1, the microarchitecture. (We will basically ignore level 3 at this point because its instructions are largely those of level 2 and the differences are not important here.) Similarly, we will call the level 2 instructions **macroinstructions**. Thus, the normal ADD, MOVE, and other instructions of the conventional machine level will be called macroinstructions. (The point of repeating this remark is that some assemblers have a facility to define assembly-time "macros" that are in no way related to what we mean by macroinstructions.) We will sometimes refer to our example level 1 machine as Mic-1 and the level 2 machine as Mac-1.

Stacks: A modern macroarchitecture should be designed with the needs of high-level languages in mind. One of the most important design issues is addressing. A mechanism must be provided for saving a current address pointer when a procedure (or function) is called and then returning back to where it came from in the calling program when exiting the procedure. In some high-level languages these called procedures are called subprograms, subroutines, or functions, and we will use these terms interchangably. A way of passing parameters to the called procedure where the called procedure will know to look for them also must be made available. The called procedure itself may need to allocate some memory space for local temporary variables in order to do its work and then be able to release the allocated space when returning to the calling program. Furthermore, a hardware mechanism that will conveniently support recursive calls (i.e., procedures calling themselves) is also desirable. Block structured languages (like Pascal and others) are normally implemented in such a way that when a procedure is exited, the storage it has been using for local variables is released. The easiest way to achieve this goal is by using a data structure called a stack.

A **stack** is a contiguous block of memory containing some data that operates on a last-in first-out basis much like a stack of cafeteria trays on a spring loaded base. A pointer (usually implemented by a CPU register) called the **stack pointer (SP)** is used to point to the current top of stack location in the region of main memory where the stack is located. Just like with the cafeteria trays, when a new tray is placed on the stack, its weight pushs down on the spring in the suporting base. Thus, stacks are sometimes called push-down stacks, and the machine instruction used to place a new data item or address on the stack is usually called a PUSH instruction. On the other hand, the instruction used to remove the top item from a stack (and place it elsewhere) is variously called by different manufacturers a POP instruction or a PULL instruction. With the cafeteria tray analogy POP likely refers to the spring in the base popping up a notch when the weight of the top tray is removed. In other contexts PULL is obviously the opposite of PUSH. In the macroarchitecture described here we will include the instructions PUSH and POP for putting data items on the stack or getting them off the stack. The register file in Fig. 5 already contains a register called SP that we can use as the stack pointer register to point to the current top of stack location in memory. It also has a PC register that we can use as a program counter to point to where the next machine instruction will be found in memory. The instruction CALL will first push the content of the PC register onto the stack before jumping off to the called procedure. The jump to the called procedure is accomplished by overwriting the PC register with a new value, called the target address (or the entry point of the procedure) and then letting the computer fetch its next instruction for execution from there. By first saving the PC register contents on the stack before overwriting the PC with a new target address, the called procedure will be able to return to the calling program where it left off. The instruction RETURN, when executed by the called procedure, will simply pop the top of stack entry into the PC register, thus pointing the program counter back to a location (the return point) in the calling program, and will in effect cause a jump back to the calling program. The CALL and RETURN instructions then provide a means for saving and then restoring the contents of the PC register using the stack when entering and exiting from called procedures.

Although one could name any register in the PUSH and POP instructions as the source of the data for a push and the destination for the data from a POP, our example machine will implicitly use only the AC register as the source of data for a PUSH and the destination for a POP. Now a PUSH must advance the stack pointer by one memory location before writing the contents of the AC register into the memory location at the top of the stack. One could choose either of the following options for how to advance the stack pointer: (1) allow the stack to grow upward from low memory addresses to high memory addresses by incrementing SP on a PUSH; or (2) allow the stack to grow downward from high memory addresses to low memory addresses by decrementing SP on a PUSH. Intel has chosen option (2) for the 80X86 architectures and so will we. Because the stack pointer points to the current top of stack location, a PUSH must first decrement (the contents of) SP and then copy the contents of the AC to the memory location whose address is in the SP register. A POP will first copy the contents of the top of stack location into the AC register and then increment (the content of) the SP register.

In order to permit programs to reserve (or delete) space on the stack for temporary local variables, instructions are needed for incrementing (or decrementing) the contents of the SP register by variable amounts. Hence, the instruction set will have instructions for incrementing SP (INSP) and decrementing SP (DESP) which allow the level 2 programmer to specify the variable amount with an 8-bit constant. Furthermore, instructions for getting at local variables or incoming parameters on the stack relative to where the SP (or some other register) currently points are also useful; thus, instructions providing a form of stack relative indexed addressing are also needed so that one doesn't have to keep moving the stack pointer to get at these items. In other words, the Mac-1 needs an addressing mode that fetches or stores a word at a known distance relative to the stack pointer (or some equivalent addressing mode). In the Mac-1 these stack pointer relative indexed addressing mode instructions will be known as load local (LODL), store local (STOL), add local (ADDL) and subtract local (SUBL); they will allow the level 2 programmer to specify a 12-bit offset (or base) value and, hence, they will have a memory reference format.

The Macroinstruction Set: The instruction set (or repertoire) is the set of all instructions that the Mac-1 is capable of executing. The Mac-1's architecture consists of a memory with 4096 16-bit words and three registers visible to the level 2 programmer. The registers are the program counter (PC), the stack pointer (SP), and the accumulator (AC) which is used for moving data around, for arithmetic, and for other purposes. Three addressing modes are provided: direct, indirect, and local. Instructions using direct addressing contain a 12-bit absolute memory address in their low-order 12 bits; and instructions using this format are usually called "memory reference instructions". Indirect addressing allows the programmer to compute a memory address, put it in the AC, and then read or write the word pointed at by the contents of the AC register; this mode is sometimes called register indirect addressing. Local addressing specifies an offset from where the SP points, and is used (among other things) to access local variables. Together, these three addressing modes provide a simple but adequate addressing system.

MAC-1 Instruction Repertoire				
OpCode Binary	**OpCode Hex**	**Assembly Mnemonic**	**Instruction**	**Meaning or Action**
`0000xxxxxxxxxxxx`	`0xxx`	`lodd`	Load direct	ac:=m[x]
`0001xxxxxxxxxxxx`	`1xxx`	`stod`	Store direct	m[x]:=ac
`0010xxxxxxxxxxxx`	`2xxx`	`addd`	Add direct	ac:=ac+m[x]
`0011xxxxxxxxxxxx`	`3xxx`	`subd`	Subtract direct	ac:=ac−m[x]
`0100xxxxxxxxxxxx`	`4xxx`	`jpos`	Jump if positive	**if** ac≥0 **then** pc:=x
`0101xxxxxxxxxxxx`	`5xxx`	`jzer`	Jump if zero	**if** ac=0 **then** pc:=x
`0110xxxxxxxxxxxx`	`6xxx`	`jump`	Jump	pc:=x
`0111xxxxxxxxxxxx`	`7xxx`	`loco`	Load constant	ac:=x (0≤x≤4095)
`1000xxxxxxxxxxxx`	`8xxx`	`lodl`	Load local	ac:=m[x+sp]
`1001xxxxxxxxxxxx`	`9xxx`	`stol`	Store local	m[x+sp]:=ac
`1010xxxxxxxxxxxx`	`axxx`	`addl`	Add local	ac:=ac+m[x+sp]
`1011xxxxxxxxxxxx`	`bxxx`	`subl`	Subtract local	ac:=ac−m[x+sp]
`1100xxxxxxxxxxxx`	`cxxx`	`jneg`	Jump if negative	**if** ac<0 **then** pc:=x
`1101xxxxxxxxxxxx`	`dxxx`	`jnze`	Jump if nonzero	**if** ac≠0 **then** pc:=x
`1110xxxxxxxxxxxx`	`exxx`	`call`	Call procedure	sp:=sp−1;m[sp]:=pc;pc:=x
`1111000000000000`	`f000`	`pshi`	Push indirect	sp:=sp−1;m[sp]:=m[ac]
`1111001000000000`	`f200`	`popi`	Pop indirect	m[ac]:=m[sp];sp:=sp+1
`1111010000000000`	`f400`	`push`	Push onto stack	sp:=sp−1;m[sp]:=ac
`1111011000000000`	`f600`	`pop`	Pop from stack	ac:=m[sp];sp:=sp+1
`1111100000000000`	`f800`	`retn`	Return	pc:=m[sp];sp:=sp+1
`1111101000000000`	`fa00`	`swap`	Swap ac, sp	tmp:=ac;ac:=sp;sp:=tmp
`11111100yyyyyyyy`	`fcyy`	`insp`	Increment sp	sp:=sp+y (0≤y≤255)
`11111110yyyyyyyy`	`feyy`	`desp`	Decrement sp	sp:=sp−y (0≤y≤255)
`1111111111111111`	`ffff`	`halt`	Halt machine	stops fetching instructions

`xxxxxxxxxxxx` is a 12-bit machine address (or constant); in column 2 it is called `xxx` and in column 5 it is called `x`.
`yyyyyyyy` is an 8-bit constant; in column 2 it is called `yy` and in column 5 it is called `y`.

Figure 11: Table of Mac-1 Instructions

The Mac-1 instruction set is shown in Fig. 11. Each instruction contains an operation code (opcode) and sometimes a memory address or constant. The opcode specifies the operation to be performed and is shown in binary in the first column of the table. The 12 x's in the instructions having a memory reference format reserve a 12-bit field for a memory address (or in the case of LOCO a constant) to be specified by the level 2 programmer. The same is true of the 8 y's in the INSP and DESP instructions that reserve an 8-bit constant field to be specified by the level 2 programmer. Column two gives the instruction encoding in hexadecimal shorthand, and column three specifies the assembly language mnemonic for each instruction's opcode. Although the assembler program for this instruction set is case sensitive and wants to see the machine instruction mnemonics in all lower-case letters, we will use upper-case in this text for emphasis when talking about specific instructions. Column four gives a short description of what the instruction does and column five specifies the action performed in a register transfer language notation. In column five, if there is more than one action occuring, then each part of the action sequence is separated from the next by a semicolon, and the sequence of actions occurs in left to right order. Column five specifies the register transfers and actions using a pseudo-Pascal language fragment. In these fragments, "m[x]" refers to memory word "x."

LODD loads the accumulator (AC register) from the memory word specified in its low-order 12 bits. LODD thus specifies direct addressing; whereas, LODL loads the accumulator from the word at a distance "x" from where the SP register points and thus specifies indexed addressing with the SP register acting as an index register. LODD, STOD, ADDD, and SUBD perform four basic functions using direct addressing, and LODL, STOL, ADDL, and SUBL perform the same functions using indexed (or local relative to the SP) addressing.

Five jump instructions are provided, one unconditional jump (JUMP) and four conditional ones (JPOS, JZER, JNEG, and JNZE). JUMP always copies its low-order 12 bits into the program counter (PC); whereas, the other four do so only if the specified condition is met.

LOCO loads a 12-bit constant in the range 0 to 4095 (inclusive) into the AC. PSHI pushes onthe the stack the word whose **address** is present in the AC register. The inverse operation is POPI, which pops a word from the stack and stores it in the memory word whose address is in the AC register. PUSHI and POPI thus specifiy register indirect addressing using the implicit AC register as the holder of the indirect address. PUSH and POP are useful for manipulating the stack in a variety of ways. SWAP exchanges the contents of AC and SP, which provides a way of loading the SP register with a new value. It is also useful for initializing SP at the start of execution. INSP and DESP are used to change SP by amounts known at compile time. Because the number of instructions to be encoded is more than a 16-bit word with a 12-bit address fields will allow, it has been necessary to tradeoff bits in the address field with bits in the op code field and use "expanding op codes" to encode all of the instructions. The offsets for INSP and DESP are limited to 8 bits in the (inclusive) range of 0 to 255. Finally, CALL calls a procedure, saving the return address on the stack, and RETN returns from a procedure by popping the return address and putting it in the PC register.

Input/Output: The Mac-1 does not have any explicit input or output instructions. Instead, it uses **memory-mapped I/O**. A read from address 4092 will yield a 16-bit word with the next ASCII character from the standard input device in the low-order 7 bits and zeros in the high-order 9 bits of the AC register. When a character is available in the data register whose address is 4092, the standard input device will set to 1 the high-order bit of the input status register at memory address 4093. The action of loading the content of the input data register at memory address 4092 into the AC register clears (i.e., sets to zero) the content of flip-flops in the status register at memory address 4093. The input routine will normally sit in a tight loop waiting for the content of 4093 to go negative. When it does, the input routine will load the AC from 4092 and return.

Output is accomplished using a similar scheme. A write (i.e., store) to the output data register at memory address 4094 copies the low-order 7 bits in the AC register to the standard output device and at the same time clears (i.e., sets to 0) the high-order bit of the output status register at memory address 4095. The high-order bit in the output status register at memory address 4095 is later set to 1 by the standard output device when it is again ready to accept another character in its data register. Standard input and output may be a terminal keyboard and visual display, or a card reader and printer, or some other combination. (Unfortunately, the simulators used to execute level 2 programs on this macroarchitecture have not as yet implemented the input/output data and status registers; so input and output are not simulated.)

An Example Microprogram

Having specified both the microarchitecture and the macroarchitecture in detail, the remaining issue is the implementation: What does a program running on the former and interpreting the latter look like, and how does it work? Here we will examine how the hardware components are controlled by the microprogram and how the microprogram interprets the conventional machine level. Early computers were not microprogrammed at all and had instructions for arithmetic, Boolean oprations, shifting, comparing, looping, and so on, that were all directly executed by the hardware. Modern day reduced instruction set computers (RISC) do likewise, but their level 2 machine instructions are merely highly encoded microinstructions; so in this case compilers translate the high level language statements into sequences of microinstructions that are easy to decode and directly control the microarchitecture's data path. Microprogrammed machines, on the other hand, interpret the level 2 machine instructions using a microprogram stored in control memory. The microprogram is written by a microprogrammer (an individual who writes microprograms and not merely a small programmer). The compilers for microprogrammed machines usually translate high-level languages into sequences of level 2 machine language statements that are in turn fetched and decoded by the microprogram that directly controls the data path's microarchitecture.

We could write the microprogram to fetch, decode and execute the level 2 machine instructions by directly specifying the sequences of 32-bit binary numbers (to be stored in control memory) that each directly control the hardware for one machine cycle comprising the four clock ticks of the four-phase cycle. This tedious task is what ultimately must be done, but having a higher level symbolic language notation that is then translated into the 32-bit numbers will make the task easier.

The Micro Assembly Language (MAL): One possible notation is to have the microprogrammer specify one microinstruction per line, naming each nonzero field and its value. For example, to add (the contents of the) AC to (the contents of the) A register and store the result in the AC register, we could write

$$\text{ENC} = 1,\ \text{C} = 1,\ \text{B} = 1,\ \text{A} = 10$$

Many microprogramming languages look like this; however, this notation is awful.

A much better idea is to use a high-level language notation, while retaining the basic concept of one source line per microinstruction. Conceivably, one could write microprograms in an ordinary high-level language, but because efficiency is crucial in microprograms, we will stick to assembly language, which we define as a symbolic language that has a one-to-one mapping onto machine instructions. Our high-level Micro Assembly Language will be called "MAL," the French word for "sick." In MAL, stores into the 16 scratchpad registers or MAR and MBR are denoted by assignment statements. Thus, the above example in MAL becomes: ac:=ac + a. (Because the intention is to make MAL Pascal-like, we adopt the usual Pascal convention of lower-case names for identifiers.)

To indicate the use of the ALU functions 0, 1, 2, and 3, we can write, for example,

ac:=a + ac, a:=band(ir,smask), ac:=a, and a:=inv(a),

respectively, where "band" stands for "Boolean AND" and "inv" stands for "invert" (i.e., bitwise logical complement). Shifts can be denoted by the functions "lshift" for left shifts and "rshift" for right shifts, as in

tir:=lshift(tir + tir)

which puts the contents of the TIR register on both the A and B buses, causes the ALU to perform an addition, and left shifts the sum 1 bit left before storing it back into the TIR register.

Unconditional jumps can be handled with **goto** statements; conditional jumps can test ALU outputs N and Z; for example,

if n **then goto** 27

Assignments and jumps can be combined on the same line. However, a slight problem arises if we wish to test a register but not make a store. How do we specify which register is to be tested? To solve this problem, we introduce the pseudo variable "alu," which can be used in the language to form a valid assignment statement but which in reality has no destination farther than the ALU's output. (Recall that the ALU is made of only combinational logic components and contains no registers or other memory devices.) For example,

alu:=tir; if n then goto 27

means that the content of the TIR register is to be run through the ALU unchanged on the A-bus (ALU code = 2) so its high-order bit can be tested. Note that this use of "alu" means that ENC = 0.

To indicate memory reads and writes, we will just put "rd" and "wr" in the source program. The order of the various parts of the source statement is, in principle, arbitrary but to enhance readability we will try to arrange them in the order that they are carried out. Fig. 12 gives a few examples of MAL statements along with the translated fields of the corresponding microinstructions (shown in decimal shorthand for each field).

Statement	AMUX	COND	ALU	SHFT	MBR	MAR	RD	WR	ENC	C	B	A	ADDR
mar:=pc; rd	0	0	2	0	0	1	1	0	0	0	0	0	00
rd	0	0	2	0	0	0	1	0	0	0	0	0	00
ir:=mbr	1	0	2	0	0	0	0	0	1	3	0	0	00
pc:=pc + 1	0	0	0	0	0	0	0	0	1	0	6	0	00
mar:=ir; mbr:=ac; wr	0	0	2	0	1	1	0	1	0	0	3	1	00
alu:=tir; if n then goto 15	0	1	2	0	0	0	0	0	0	0	0	4	15
ac:=inv(mbr)	1	0	3	0	0	0	0	0	1	1	0	0	00
tir:=lshift(tir); if n then goto 25	0	1	2	2	0	0	0	0	1	4	0	4	25
alu:=ac; if z then goto 22	0	2	2	0	0	0	0	0	0	0	0	1	22
ac:=band(ir, amask); goto 0	0	3	1	0	0	0	0	0	1	1	8	3	00
sp:=sp + (-1); rd	0	0	0	0	0	0	1	0	1	2	2	7	00
tir:=lshift(ir + ir); if n then goto 69	0	1	0	2	0	0	0	0	1	4	3	3	69

Figure 12: Some MAL statements and their corresponding microinstructions.

The Example Microprogram: We have finally reached the point where we can put all the pieces together. Fig. 13 is the microprogram that runs on the Mic-1 and interprets the Mac-1. It is a surprisingly short program – only 81 lines. By now the choice of names for the scratchpad registers in Fig. 5 is obvious: PC, AC, and SP are used to hold the three Mac-1 registers. IR is the instruction register and holds the macroinstruction currently being executed. TIR is a temporary copy of the IR, used for decoding the opcode. The next three registers hold the indicated constants. AMASK is the address mask $0FFF_{16}$, and is used to separate out opcode and address bits. SMASK is the stack mask, $00FF_{16}$, and is used in the INSP and DESP instructions to isolate the 8-bit offset value. The remaining six registers have no assigned function and can be used as scratch registers for whatever the microprogrammer wishes.

Like all interpreters, the microprogram in Fig. 13 has a main loop that fetches, decodes, and executes instructiions from the program being interpreted, in this case level 2 instructions. Its main loop begins on line 0, where it begins fetching the macroinstruction whose memory address is in the PC register. While waiting for this instruction to arrive, the microprogram increments the content of the PC and continues to assert the "Rd" bus signal. When it arrives, in line 2, it is stored in the IR register and simultaneously the high-order bit (bit 15) is tested. If bit 15 is a 1, decoding proceeds to line 28; otherwise, it continues on line 3. Assuming for the moment that the instruction is a LODD, bit 14 is tested on line 3, and the TIR register is loaded with the original instruction shifted left 2 bit positions, one shift using the adder and one using the shifter. Note that the ALU status bit N is determined by the ALU output in which bit 14 is the high-order bit, because IR + IR shifts the IR contents left 1 bit position. The shifter output does not affect the ALU status bit.

All instructions having 00 in their two high-order bits eventually come to line 4 to have bit 13 tested, with the instructions beginning with 000 going to line 5 and those beginning with 001 going to line 11. Line 5 is an example of a microinstruction with ENC = 0; it just tests the content of the TIR register, but does not change it. Depending on the outcome of this test, the code for LODD or STOD is selected.

For LODD, the microcode must first fetch the word directly addressed by loading the low-order 12 bits of the IR into the MAR. In this case, the high-order 4 bits are all zero, but for STOD and other instructions they are not. However, because the MAR is only 12 bits wide and connected to only the low-order 12 bits on the B-bus, the opcode bits do not affect the choice of the word to be read. In line 7, the microprogram has nothing to do, so it just waits. When the word arrives, it is copied into the AC register and the microprogram jumps back to the top of the loop where the instruction fetch cycle begins. STOD, ADDD, and SUBD are similar. The only noteworthy point concerning them is how subtraction is done.

Recall that in radix r the radix complement (RC) of a number x is defined to be $\mathrm{RC}(x) = r^n - x$. Similarly, the diminished radix complement (DRC) of x (also called the $r-1$'s complement) is defined to be $\mathrm{DRC}(x) = r^n - r^{-m} - x$. When $m = 0$ so that we are dealing only with n-bit registers containing integers, then the 1's complement of x is $1\text{'s}(x) = 2^n - 2^0 - x = 2^n - 1 - x$. The 2's complement of x is then $2\text{'s}(x) = 2^n - x = 1'\mathrm{s}(x) + 1$, where the 1's complement of x is the same as the bitwise logical complement of the n-bit number x. Thus, SUBD makes use of the fact that

$$x - y = x + (-y) = x + (\overline{y} + 1) = x + 1 + \overline{y}$$

in two's complement. The addition of 1 to the content of the AC is done on line 16 (using the commutativity of additiion); otherwise line 16 would be wasted like line 13.

The microcode for JPOS begins on line 21. If the content of the AC < 0, the branch fails and JPOS is terminated immediately by jumping back to the main loop and fetching the next instruction in sequence. If, however, the content of the AC ≥ 0, the low-order 12 bits of the IR are extracted by ANDing them with the $0\mathrm{FFF}_{16}$ mask in the AMASK register and storing the result in the PC register. It does not cost anything extra to remove the opcode bits here, so we might as well do it. If it had cost an extra microinstruction, however, we would have had to look very carefully to see if having garbage in the high-order 4 bits of the PC could cause trouble later.

In a certain sense, JZER (line 23) works the opposite of JPOS. With JPOS, if the test condition is met, the jump fails and control returns to the main loop. With JZER, if the test condition is met, the jump is taken. Because the code for performing the jump is the same for all jump instructions, we can save microcode by just going to line 22 whenever feasible. This style of programming generally would be considered uncouth in an application program, but in a microprogram no holds are barred. Performance is everything.

Microprogram to fetch, decode, and execute Mac-1 instructions			
Adr: Microinstruction	Comment	Adr: Microinstruction	Comment
0: mar:=pc; rd;	**fetch instr**	41: alu:=tir; if n then goto 44;	decode ir_{12}
1: pc:=pc + 1; rd;	increment pc	**42**: alu:=ac; if n then goto 22;	1100 = **JNEG**
2: ir:=mbr; if n then goto 28;	decode ir_{15}	43: goto 0;	
3: tir:=lshift(ir + ir); if n then goto 19;	decode ir_{14}	**44**: alu:=ac; if z then goto 0;	1101 = **JNZE**
4: tir:=lshift(tir); if n then goto 11;	decode ir_{13}	45: pc:=band(ir,amask); goto 0;	
5: alu:=tir; if n then goto 9;	decode ir_{12}	46: tir:=lshift(tir); if n then goto 50;	decode ir_{12}
6: mar:=ir; rd;	0000 =**LODD**	**47**: sp:=sp + (-1);	1110 = **CALL**
7: rd;		48: mar:=sp; mbr:=pc; wr;	
8: ac:=mbr; goto 0;		49: pc:=band(ir,amask); wr; goto 0;	
9: mar:=ir; mbr:=ac; wr;	0001 = **STOD**	50: tir:=lshift(tir); if n then goto 65;	decode ir_{11}
10: wr; goto 0;		51: tir:=lshift(tir); if n then goto 59;	decode ir_{10}
11: alu:=tir; if n then goto 15;	decode ir_{12}	52: alu:=tir; if n then goto 56;	decode ir_{9}
12: mar:=ir; rd;	0010 = **ADDD**	**53**: mar:=ac; rd;	1111-0000 = **PSHI**
13: rd;		54: sp:=sp + (-1); rd;	
14: ac:=mbr + ac; goto 0;		55: mar:=sp; wr; goto 10;	
15: mar:=ir; rd;	0011 = **SUBD**	**56**: mar:=sp; sp:=sp + 1; rd;	1111-0010 = **POPI**
16: ac:=ac + 1; rd;		57: rd;	
17: a:=inv(mbr);		58: mar:=ac; wr; goto 10;	
18: ac:=ac + a; goto 0;		59: alu:=tir; if n then goto 62;	decode ir_{9}
19: tir:=lshift(tir); if n then goto 25;	decode ir_{13}	**60**: sp:=sp + (-1);	1111-0100 = **PUSH**
20: alu:=tir; if n then goto 23;	decode ir_{12}	61: mar:=sp; mbr:=ac; wr; goto 10;	
21: alu:=ac; if n then goto 0;	0100 = **JPOS**	**62**: mar:=sp; sp:=sp + 1; rd;	1111-0110 = **POP**
22: pc:=band(ir,amask); goto 0;	perform jump	63: rd;	
23: alu:=ac; if z then goto 22;	0101 = **JZER**	64: ac:=mbr; goto 0;	
24: goto 0;	else don't jump	65: tir:=lshift(tir); if n then goto 73;	decode ir_{10}
25: alu:=tir; if n then goto 27;	decode ir_{12}	66: alu:=tir; if n then goto 70;	decode ir_{9}
26: pc:=band(ir,amask); goto 0;	0110 = **JUMP**	**67**: mar:=sp; sp:=sp + 1; rd;	1111-1000 = **RETN**
27: ac:=band(ir,amask); goto 0;	0111 = **LOCO**	68: rd;	
28: tir:=lshift(ir + ir); if n then goto 40;	decode ir_{14}	69: pc:=mbr; goto 0;	
29: tir:=lshift(tir); if n then goto 35;	decode ir_{13}	**70**: a:=ac;	1111-1010 = **SWAP**
30: alu:=tir; if n then goto 33;	decode ir_{12}	71: ac:=sp;	
31: a:=ir + sp;	1000 = **LODL**	72: sp:=a; goto 0;	
32: mar:=a; rd; goto 7;		73: tir:=lshift(tir); if n then goto 76;	decode ir_{9}
33: a:=ir + sp;	1001 = **STOL**	**74**: a:=band(ir,smask);	1111-1100 = **INSP**
34: mar:=a; mbr:=ac; wr; goto 10;		75: sp:=sp + a; goto 0;	
35: alu:=tir; if n then goto 38;	decode ir_{12}	76: alu:=tir; if n then goto 80;	decode ir_{8}
36: a:=ir + sp;	1010 = **ADDL**	**77**: a:=band(ir, smask);	1111-1110 = **DESP**
37: mar:=a; rd; goto 13;		78: a:=inv(a);	
38: a:=ir + sp;	1011 = **SUBL**	79: a:=a + 1; goto 75;	
39: mar:=a; rd; goto 16 ;		**80**: halt; goto 80;	1111-1111 = **HALT**
40: tir:=lshift(tir); if n then goto 46;	decode ir_{13}		

The execution cycle for each decoded MAC-1 instruction begins at the control store address whose line is labeled with a comment showing the assembly language mnemonic for the corresponding instruction (capitalized for emphasis). "Adr:" is the control store address. The instruction fetch cycle begins at control store address zero.

Figure 13: Microinstructions to fetch, decode, and execute Mac-1 instructions on the example Mic-1 microarchitecture

JUMP and LOCO are straightforward, so the next interesting execution routine is for LODL. First the absolute memory address to be referenced is computed by adding the offset contained in the instruction to the content of the SP register. Then the memory read is initiated. Because the rest of the code is the same for LODL and LODD, we might as well use lines 7 and 8 for both of them. Not only does this save control store space with no loss of execution speed but it also means fewer routines to debug. Analogous code is used for STOL, ADDL, and SUBL. The code for JNEG and JNZE is similar to JZER and JPOS, respectively (not the other way around). CALL first decrements the content of the SP register, then pushes the return address (which is the current content of the PC register) onto the stack, and finally jumps to the called procedure. Line 49 is almost identical to line 22; if it had been exactly the same, we could have eliminated line 49 by putting an unconditional jump to 22 in 48. Unfortunately, we must continue to assert "Wr" for another microinstruction.

The rest of the macroinstructions all have 1111 as their high-order 4 bits, so decoding of (at least some of) the low-order 12 bits in these instructions is required to tell them apart. The actual execution routines are straightforward so we will not comment on them further.

A few more points are worth making. In Fig. 13 we increment the content of the PC register in line 1. It could equally well have been done in line 0, thus freeing line 1 for something else while waiting for memory to respond. In this machine there is nothing else to do, but in a real machine the microprogram might use this opportunity to check for I/O devices awaiting service, refresh dynamic RAM, or something else.

If we leave line 1 the way it is , however, we could speed up the machine by modifying line 8 to read

mar:= pc; ac:= mbr; rd; goto 1;

In other words, we can start fetching the next instruction before we have really finished with the current one. This capability provides a primitive form of instruction pipelining. The same trick can be applied to other execution routines as well.

It is clear that a substantial amount of the execution time of each macroinstruction is devoted to decoding it bit by bit. This observation suggests that it might be useful to be able to load the MPC register under microprogram control. On many existing computers the microarchitecture has hardware support for extracting macroinstruction opcodes and stuffing them directly into the MPC to effect a multiway branch. If, for example, we could shift the IR 9 bits to the right and put the resulting number into the MPC, we would have a 128-way branch to locations 0 through 127. Each of these words would contain the first microinstruction in the execution sequence for the corresponding macroinstruction. Although this approach wastes control store space, it greatly speeds up the machine, so something like it is nearly always used in practice.

By using memory-mapped I/O, the CPU is not aware of the difference between true memory addresses and I/O device registers. The microprogram handles reads and writes to the top four words of the address space the same way it handles any other reads and writes.

Designing a machine as a series of levels is done for efficiency and simplicity because each level deals only with another level of abstraction. The level 0 designer worries about how to squeeze the last few nanoseconds out of the ALU by using some means to reduce carry-propagation time. The microprogrammer worries about how to get the most mileage out of each microinstruction, typically by exploiting as much of the hardware's inherent parallelism as possible. The macroinstruction set designer worries about how to provide an interface that both the compiler writer and microprogrammer can learn to love, and be efficient at the same time. Clearly, each level has different goals, problems, techniques, and in general, a different way of looking at the machine. By splitting the total machine design problem into several subproblems, we can attempt to master the inherent complexity in designing a modern computer.