INTRODUCTION TO BIT SLICES AND MICROPROGRAMMING

Andries van Dam
Brown University, Providence, Rhode-Island, USA

Abstract

Bit-slice logic blocks are fourth-generation LSI components which are natural extensions of traditional multiplexers, registers, decoders, counters, ALUs, etc. Their functionality is controlled by microprogramming, typically to implement CPUs and peripheral controllers where both speed and easy programmability are required for flexibility, ease of implementation and debugging, etc. Processors built from bit-slice logic give the designer an alternative for approaching the programmability of traditional fixed-instruction-set microprocessors with a speed closer to that of hardwired "random" logic.

Introduction

The purpose of this set of annotated lecture transparencies is to give a brief introduction to the use of bit-slice logic in microprogrammed engines (CPUs) and controllers. A basic understanding of the goals of the technology and its potential will allow one to read the literature with some idea of what the important issues and design parameters might be. Bit slices will be placed in the spectrum of hardware/software building blocks, and their basic types and uses will be briefly illustrated. Since slices are controlled typically by microprograms, an elementary review of that subject will also be given, especially to stress the crucial point that working with bit slices requires a proper (and integrated) understanding of hardware, firmware and software, as well as the use of proper tools and methodologies for each of these levels of design.

The reader is referred to Glenford J. Myers' excellent brand-new book Digital System Design with LSI Bit-Slice Logic (Wiley-Interscience, 1980) for a full treatment, to Prof. V. Zacharov's lecture notes on technology in these Proceedings for a review of the generations of digital building blocks, and to Dr. C. Halatsis' lecture notes on software for microprocessors, also in these Proceedings, for a discussion of tools and methodology for working with bit slices. As a relevant example, the MICE PDP-11 fixed-point engine used for online data filtering was built at CERN using bit-slice technology, and is described in References 1 and 2.

Review of Conventional CPU Design

Figure 1 is a high-level block diagram of a conventional CPU, showing both processing (data transformation in the APU) and control (affecting data transfers between components, registers, and activations of functional units such as the ALU, shifter, etc.). Figure 2 shows a typical MOS fixed-instruction-set microprocessor implementation of such a CPU, with a CPU chip for most CPU functions (including the instruction cycle-fetch, increment, execute, ALU functions, I/O functions for both memories and devices and control of other chips), a bus control chip for arbitrating and controlling the three standard busses (address, data, and control), the clock chip to provide timing pulses/waveforms to drive all other chips, and an Interrupt Control Unit chip to provide a priority interrupt mechanism. Note that memory and device controller chips are handled as much as possible in the same way, each connected to the three standard busses.

Bit Slices

With current MOS (V)LSI technology, it is possible to pack all of the CPU functionality for an 8-bit or even a 16-bit processor in one or a few chips, where the limitation is one of pin-out - 64 pins is a reasonable upper limit with today's state of the art. Given the traditional speed/power/packing density tradeoffs between MOS and bipolar, the obvious question is whether a 64-pin bipolar microprocessor for an 8-bit or 16-bit processor is feasible. The answer, not surprisingly, is no, because of packing density. What then is the right functionality to assign to bipolar chips? Figure 3 shows a simple functional decomposition of a CPU, where one or more function rows are represented by each chip of a traditional MOS microprocessor. Bipolar bit slices, because of their lesser packing density, cannot even take care of an entire 16-bit row's functionality, so they only do a 4-bit slice worth (shown in the vertical ruling), but in such a way that slices can be chained together, four for example, to give 16-bit functionality. (Some families may provide only 2-bit slices and others may include 8-bit building blocks, particularly for memories.) This technique should be viewed simply as a natural extension of using well-known (4-bit) digital building blocks for making arbitrary-size (de)multiplexors, registers, counters, shifters, (R)ALUs (register file plus ALU), etc.

On the spectrum of hardware/software building blocks shown in Figure 4, it is clear that bit slices are bipolar LSI hardware chips providing a level of functionality between that of MSI random-logic components and programmable MOS microprocessor CPU chips, but much closer to MSI components! It is, in fact, a misnomer to speak of bit-slice

microprocessor chips since they can only be used as building blocks in a "microprocessor" CPU design, in the same way in which standard components such as RALUs are. Bit slices are hardware, not software, but require external microprogrammed control to regulate their functionality, as discussed below.

To put it another way, bipolar LSI technology, coupled with microprogramming as the control technique, have resulted in a new medium for digital design which gives an excellent compromise between the speed of random (hardwired) bipolar logic and the flexibility of the programmable microprocessor CPU chip. Cycle times for bit-slice designs vary from roughly 100 to 250 nsec (compared to 15-100 for the fastest ECL SSI designs). While not as fast as discrete hardware, bit-slice processors are also not as easy to program as conventional microprocessors because, as we shall see, microprogramming involves far more intimate knowledge of hardware than does even assembly-language programming for a conventional microprocessor. Nonetheless, the speed advantage of a factor of typically 2-5 over such MOS microprocessors may often allow the use of bit slices where previously a (non-programmable) hardwired design would have been required. Furthermore, bit slices, by their very nature, allow processor width appropriate to the problem, thereby avoiding the time multiplexing of busses of conventional microprocessors whose widths are often too small.

In Figure 5 we see a high-level diagram of a conventional 4-bit ALU slice making a 16-bit ALU. Each ALU slice communicates with three data busses, two inputs (A and B) and one output (O); the 16-bit operands are split so that the high-order bits (0-3) are handled by the high-order slice, 4-7 by the second slice, etc. All four slices operate in parallel under the control of the 7-bit shared opcode bus which has them all execute the same ALU function. (Only the control of carry-in and carry-out may differ in first and last stages.) Status is propagated in the normal fashion between stages, resulting in 16-bit status at the output of the high-order stage. Also, for ALU slices, look-ahead carry logic is typically available.

In addition to ALU slices (with or without built-in register file), manufacturer families typically support a sequencer (or microprogram controller) slice for controlling all other slices (and additional MSI/SSI logic such as multiplexors and registers), external register file slice, priority interrupt controller slice, DMA controller slice, memory interface slice, etc., all of which are compatible with conventional logic and memories in the same technology. The most important (R)ALU and sequencer slices are discussed in more detail below.

A typical design for a bit-slice-based processor is shown in Figure 6. Note the 16-bit arithmetic section made

from 4 RALU slices, whose status bits are fed, along with other status bits, to the control section. A 12-bit, 3-slice microsequencer fetches microinstruction control words from the microprogram control store and stores each for execution in a pipeline register, in effect a microinstruction register. Bits in the control word are used to condition ("program") each of the slices, busses, and other (discrete) logic in the processor to carry out the selected functionality for the duration of the microcycle. Again, this microprogrammed control will be explained in more detail below.

A slightly more detailed design is shown in Figure 7, which is based on the Motorola 10800 family of slices. A 16-bit ALU is made from 4 ALU slices plus look ahead carry chips, taking operands from an external register file made from two 8-bit dual-ported slices. An 8-bit, 2-slice sequencer receives the low-order bits from the IB and OB busses and reads from 256 locations of control store. Some of the individual fields of the microinstruction are shown in the pipeline register. Finally, a 16-bit memory interface, using 4 slices, controls an external memory and provides addressing via effective address arithmetic using address data from the register file.

A data filtering engine designed at CERN DD which can be microprogrammed directly or programmed in PDP 11 assembler or Fortran via an emulator for the PDP 11 (i.e., a microprogrammed interpreter) is shown in Figure 8. At this level of detail it is very similar to the previous design, using a 12-bit sequencer and a 118 bit-wide microinstruction/control store format. Note again the division of the microinstruction into fields controlling both the slices and the additional logic required to glue the slices together (such as the target instruction decoding ROMs).

This glue is added to the basic bit slices both to provide simple functionality which they don't have (e.g. additional 4-bit multiplexor "slices", registers, counters, memories, etc. preexisting in the logic family) and to bypass functionality of a slice which is too slow to be utilized. Thus the auxiliary address control is sometimes used to bypass the 10803 memory interface address calculation when to use it would require an extra cycle. Similarly, the target instruction decoding ROM gives single-cycle mapping of a PDP 11 target instruction address mode to the microsubroutine to do the corresponding address calculation, bypassing the multi-microcycle functionality of the 10801 sequencer to do the next microaddress calculation.

This points out that the designer need not, indeed cannot use all the functionality of a slice, and must supplement it with standard components to provide missing functionality or to bypass functionality that is too slow. On

the average, however, most functions can be executed by the slices if one studies the data sheets very carefully to see how the many data and control functions can be marshalled. This is no small task, given the often-cryptic and idiosyncratic descriptions on the sheets; what's worse, slices are notoriously unstructured and asymmetrical (unorthogonal) - only certain combinations of data flows and internal operations work, for reasons typically left unspecified. Experimentation may be required to find out exactly what a slice is capable of!

Review of Microprogramming

Each bit slice has typically one or more (4-bit) operand data pins and from 5 to 20 control pins which condition its data and control paths/options/functions, much in the way that a classical ALU has two 4-bit source operands, a 4-bit destination operand, and a 5-bit opcode to select among 16 arithmetic and 16 logic functions to be applied to the sources (and destination). Rather than getting lost in the details of controlling each individual pin on each slice, let's step back a moment and look at the problem of control in digital computers.

Figure 9 shows a finite-state graph representation of the fetch-increment-execute instruction cycle, with much detail hidden by the use of macro states. A transition is made from $State_i$ to $State_j$ as a function of input, either a clock pulse or a clocked/strobed data/control input. States can be implemented via registers, transitions by changing the contents of those registers. Any finite-state graph can be implemented as a collection of flip-flops/registers and combinatorial logic controlling the inputs to the registers (using a synthesis technique not needed here).

What is important to us is the notion that all actions in a computer can be divided into source -> destination data flows (register transfers) and activations of processing (data transformation) functions such as ALU or shifter units. The latter process can be symbolized as a +V -> processing unit "transfer" (Figure 10a). As with the state graph formalism, the trick is to be complete in the enumeration: itemize all states and transitions in and out of them; itemize all registers, processing units, data paths and control paths, and show which is active under which conditions (Figure 10b). In Figure 10c we show a method of synthesizing control of an individual source -> destination register transfer. The control network enabling both the data flow and its strobing into the destination flip-flop is simply an enumeration of the form: "if it's an add instruction (corresponding to a certain bit pattern in the opcode bits of the instruction register i_0, i_1, ..., i_7), and it's clock pulse 2 and major phase 3 or if it's the subtract instruction...".

In its most elementary form, then, the design of a digital network such as a controller or CPU consists of enumerating all registers, functional units, paths, and their control networks. Wilkes realized as early as 1951 that the control networks were the essence of the design of the control, and that they could be collected in a diode matrix ROM to provide both an orderly design procedure and an orderly overview of the result (Figure 11). Calling the set of control actions (enablings of transfers and activations) at each clock pulse a microinstruction, the collection of diodes in a row of the matrix is its binary representation, and a sequence of microinstructions carries out the sequences of transfers/activations which correspond to a target-level or macrolevel instruction. To be more precise, the static target opcode (e.g., PDP 11 add register, indexed storage instruction) is mapped to a sequence of microinstructions, each of which (in Wilkes' design) specifies the micro address of its successor, and which in concert do opcode decoding, address calculation/operand fetch, and finally opcode execution using the ALU followed by destination loading. Each bit in each microinstruction controls a single resource/register transfer/activation, and we see that typically multiple activities take place during each microinstruction cycle.

In modern implementations, the diode matrix becomes a (writable) high-speed control store, and individual bits still control individual gates or, grouped in fields, they control the operations of combinatorial and sequential logic such as multiplexors, ALU slices, sequencer slices, etc.

It should be noted that microprogrammed control allows easy changes in the control of the basic hardware components but does involve a typical instruction fetch, decode, increment/next microaddress calculation cycle, which takes time. Pipelining the instruction fetch and execution via a microinstruction pipeline register cuts down some of this time, but causes the problem that conditional branching tests and status of the ALU are set during a previous microinstruction, something the microprogrammer needs to be conscious of.

Reviewing some other microprogramming terminology, we see that the microprogram often is used to create a virtual target machine by interpreting its instruction set - this interpreter, stored in control store ("firmware") is called the emulator. Emulation allows an arbitrary host microarchitecture to simulate a standard target architecture such as the IBM 370 or the PDP 11 to achieve a family of dissimilar processors, each with a different technology, host architecture, and price/performance ratio, but the instruction set, and therefore the software, are compatible. Conversely, the microprogram may simply implement a standard algorithm such as an FFT or peripheral controller to create

a fixed-function, nonprogrammble black box. In the case of MICE, one can control the lowest-level hardware by microprogramming for maximum speed (after adding special-purpose hardware, of course) but at the cost of considerable hardware knowledge and idiosyncratic (micro) assembler programming. A much simpler but slower form of programming can be done using the PDP 11 emulator, either in standard PDP 11 assembler or Fortran. Algorithms requiring speed can be directly microcoded and invoked as extended target instructions from normal PDP 11 code. Floating-point accelerators or Fortran assists are commercial implementations of this notion.

The next bit of terminology is horizontal versus vertical microprogramming. Wilkes' model of one line/resource is fully horizontal (Figure 13a) in that the control word is as wide as the number of individual resources to control (potentially many hundreds of bits). More typically, bits are grouped and then decoded with control slices, random logic, and with on-slice decoding. Multiple-instruction formats such as those encountered with "normal" target architectures result (Figure 13b), and several vertical, compressed instructions may be required to implement the equivalent of one wide horizontal instruction. Note that the horizontal instruction allows potentially greater parallelism and is faster, since there is no decoding - it is, however, a great deal messier to "program". Vertical instruction formats often have a three-address structure, operating on two register source operands to produce a third register destination operand. Simultaneously, memory read/write operations and completion tests may be performed.

Note, then, several characteristics of microprogramming:

1. It is control of the lowest-level hardware components, involving a thorough understanding of timing;

2. It is the lowest-level "bit-flicking" available;

3. The host architectures are frequently very complex, with many data and control flows possible - emulators create more "reasonable" target architectures;

4. Target memory read/write is effectively I/O which must be carefully timed;

5. Code for conventional microprocessors stored in ROMs is erroneously called "firmware" and has nothing to do with microprogramming.

Figure 14 shows again some distinctions between hardware, firmware and software implementations: firmware affords the best tradeoff for many cases between speed and programming flexibility for ease of design, alteration, etc.

Microprogramming is like assembly language programming in that algorithms and top-down structured design are paramount, as is the use of proper design, implementation, debugging and documentation tools. It is also like hardware design in that bottom-up hardware considerations consistently intrude (parallelism, critical races, bus conflicts, timing). For the experienced designer it is an extremely effective tool, and is the only way to take advantage of the bit slice medium.

An Overview of AMD and Motorola Bit Slices

The AMD 2900 family and Motorola 10800 family represent the current state of the art in bit-slice logic. The 2900 series logic, based primarily on LSTTL technology, has the largest variety of devices of any available bit-slice processor family. At the other end of the bit-slice spectrum, the 10800 series logic utilizes ECL technology and offers the fastest cycle times. A brief overview of each family follows.

AMD has defined eight fundamental system functions that the 2900 family is to support:

1. Data manipulation;
2. Microprogram control;
3. Macroprogram control;
4. Priority interrupt;
5. Direct memory access;
6. I/O control;
7. Memory control;
8. Front panel control.

There are over 50 device types to choose from, providing the designer with a great deal of flexibility in tailoring his system.

The most popular bit-slice device is the 2901 RALU, shown in Figure 15. Each slice is 4 bits wide, and any number of 2901's can be connected together for longer word lengths. The eight-function ALU performs addition, two subtraction operations, and five logic functions. The 16 registers are stored in a two-port RAM, enabling simultaneous access to two working registers. The same basic architecture is maintained in two other RALU devices, the 2903 and 29203, which have added features such as built-in multiplication and division logic. An example of a microprogram

sequencer, the 2909, is shown in Figure 16. The internal push-pop stack is four words deep and is used to nest subroutines. The 2909 can select its output address from any one of four sources:

1. the stack;

2. the program counter;

3. an internal register;

4. an external direct input.

The slice is 4 bits wide and cascadable to any number of microwords. Some of the other more popular devices include the 2914 Vectored Priority Interrupt Controller, and 2930 Program Control Unit and the 2940 DMA Address Generator.

In contrast to the rich assortment of parts in the 2900 family, there are only a few in the 10800 series. They include the 10800 ALU, the 10801 Microsequencer Control, the 10803 Memory Interfact Function, the 10806 Dual Access Stack, and the 10808 Shifter Function. Although primitive, the 10800 family is very fast. The 10800 ALU, for example, can operate at a 15 MHZ clock rate, which represents a cycle time of 60ns.

The 10800 ALU is shown in Figure 17. Like the 2901 ALU, it is a 4-bit slice and fully expandable to larger word lengths by connecting devices in parallel. The significant difference between the 2900 and 10800 architecture is the lack of an internal register file in the 10800. The 10806 Dual Access Stack is designed to perform this function externally. The 10801 Microsequence slice is shown in Figure 18. Similar to the 2909, the 10801 has an internal stack for nesting subroutines and is 4 words deep. The 10803 Memory Interface slice is shown in Figure 19. This device generates memory addresses and provides for the bidirectional transfer of processor data. It contains both the MAR (Memory Address Register) and the MDR (Memory Data Register). The ALU in the 10803 has 7 functions; addition, subtraction, and 5 logic operations. Unlike the 10800, the 10803 maintains an internal register file consisting of four words.

MICE as an Example

Figure 20 is a more detailed version of the block diagram of Figure 8, shown to illustrate the separate number of fields and which chips/gates they control (in balloons). Typical flows are to do a three-address ALU operation on the register file, while doing a target memory fetch, and determining the next microaddress. Pipelining at

both the micro level and the emulated target level is employed to give overlap speed for combinations of operations that don't cause bus or slice contention.

References

1. Barbacci, M., Halatsis, C., Joosten, J., Letheren, M., van Dam, A., "Simulation of a Horizontal Bit Sliced Processor Using the ISPS Architecture Simulation Facility", IEEE Computers (special issue on firmware engineering), February, 1981.

2. Halatsis, C., Joosten, J., Letheren, M., van Dam, A., "Architectural Considerations for a Microprogrammable Emulating Engine Using Bit-Slices", Proceedings 7th International Symposium on Computer Architecture La Baule, France, May, 1980.

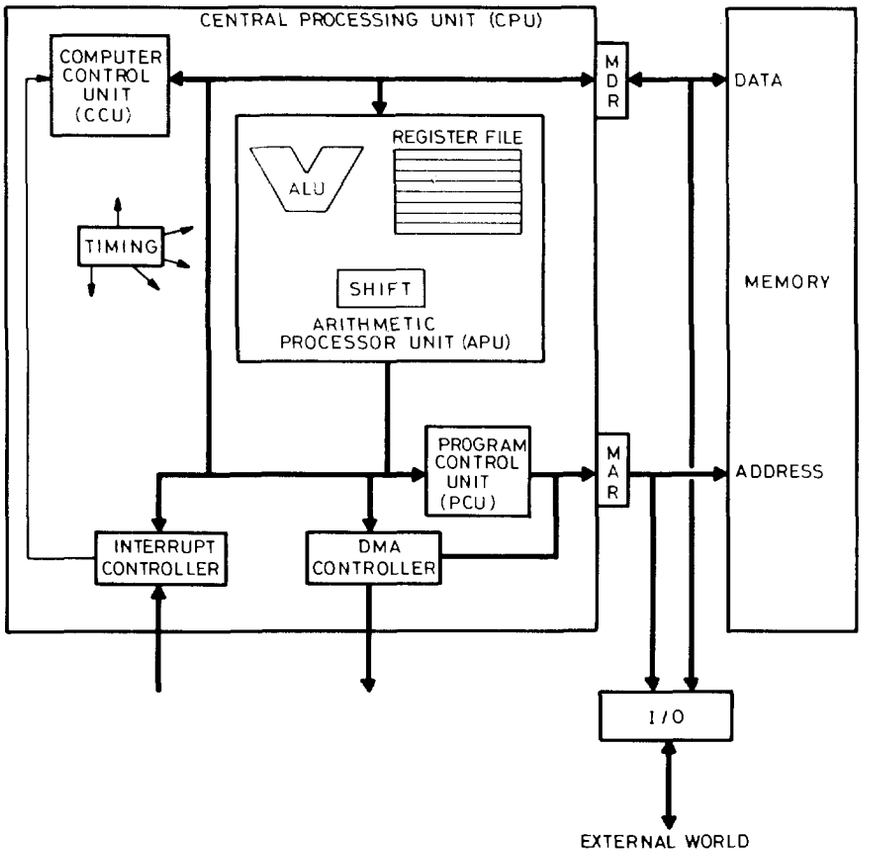


Fig. 1 Conventional CPU and connections to other components

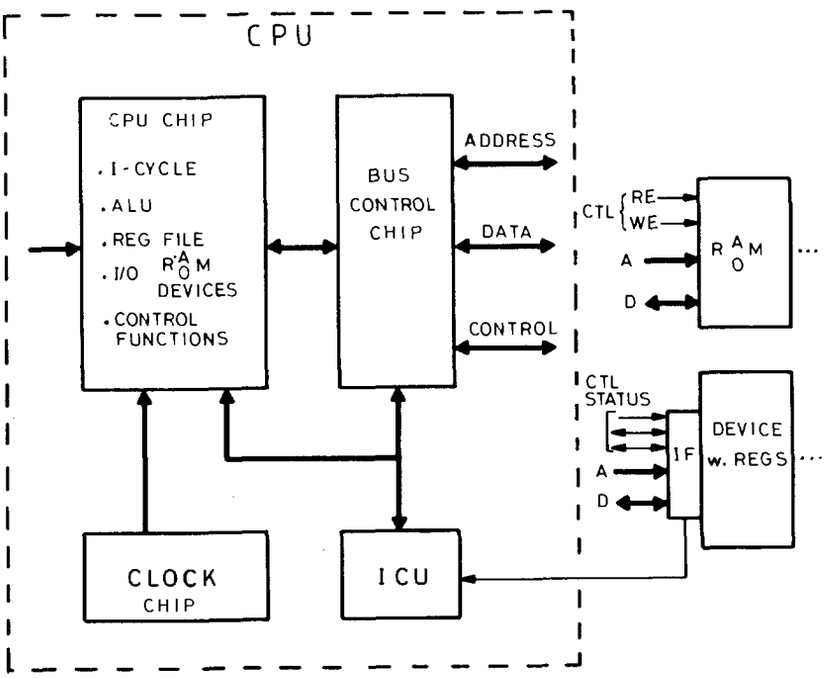


Fig. 2 High-level logical view of typical MOS, fixed instruction set microprocessor used as microcomputer.

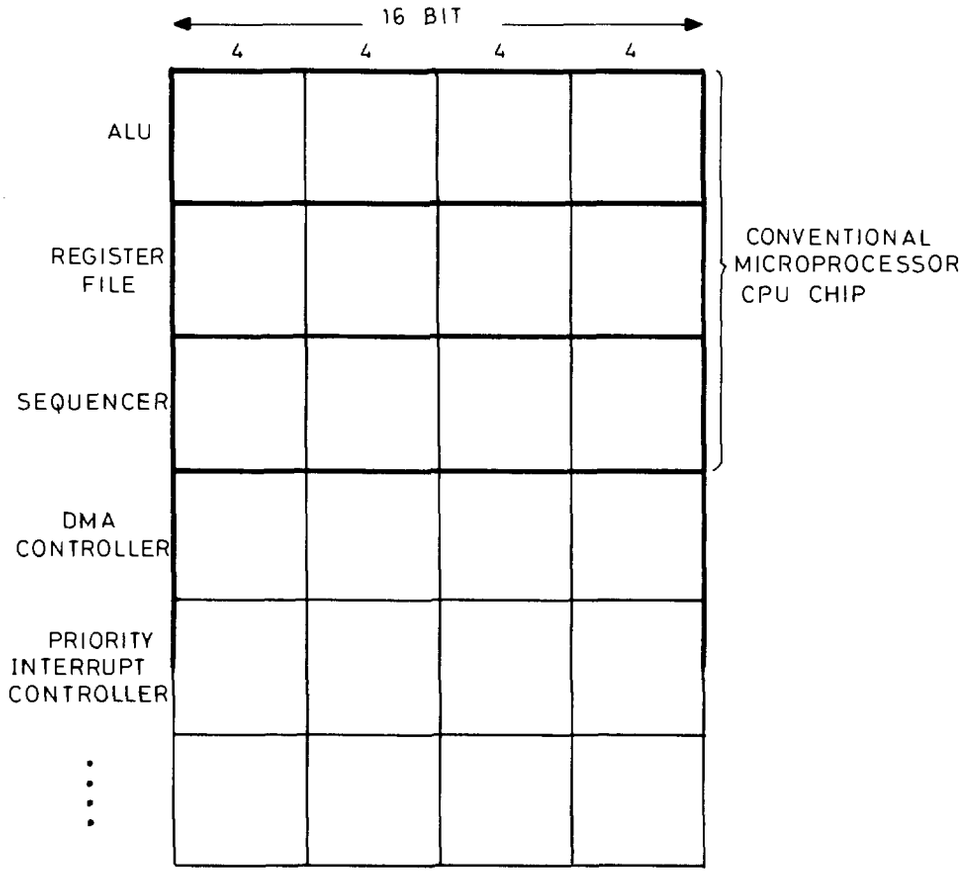


Fig. 3 Functional decomposition of 16-bit CPU

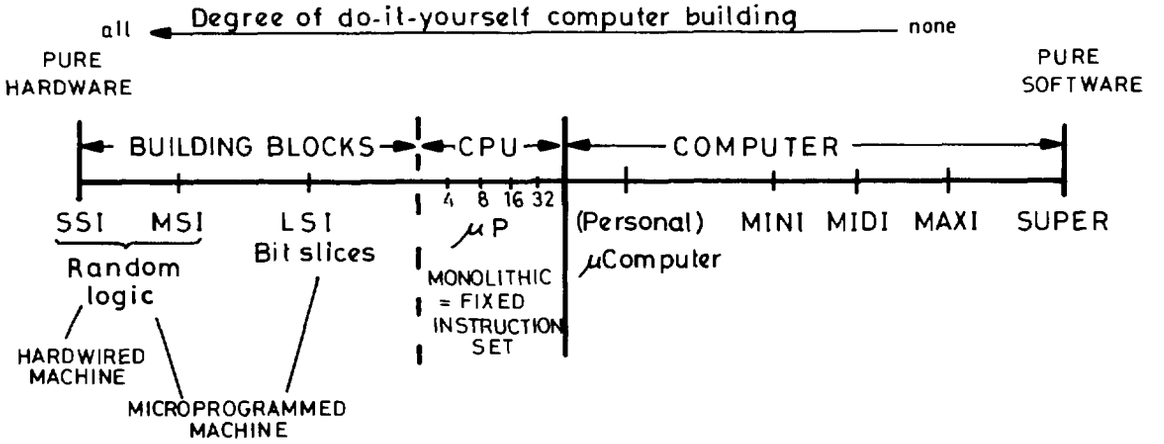


Fig. 4 Bit slices on the building block spectrum

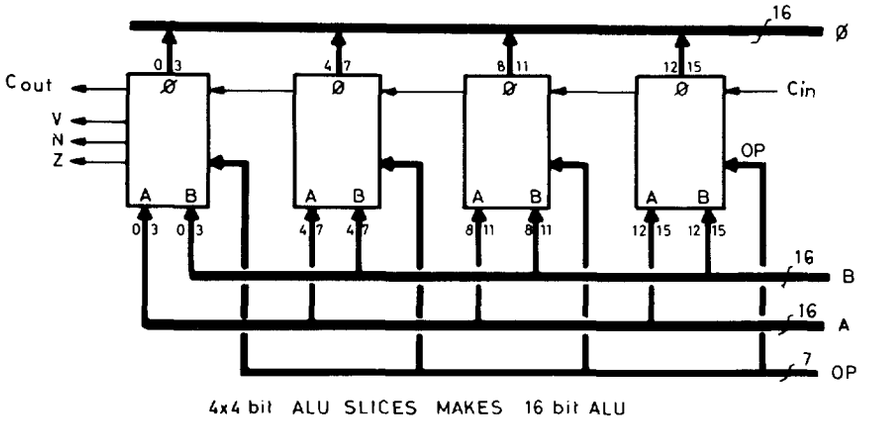


Fig. 5 Slices share control lines, split data buses, propagate status.

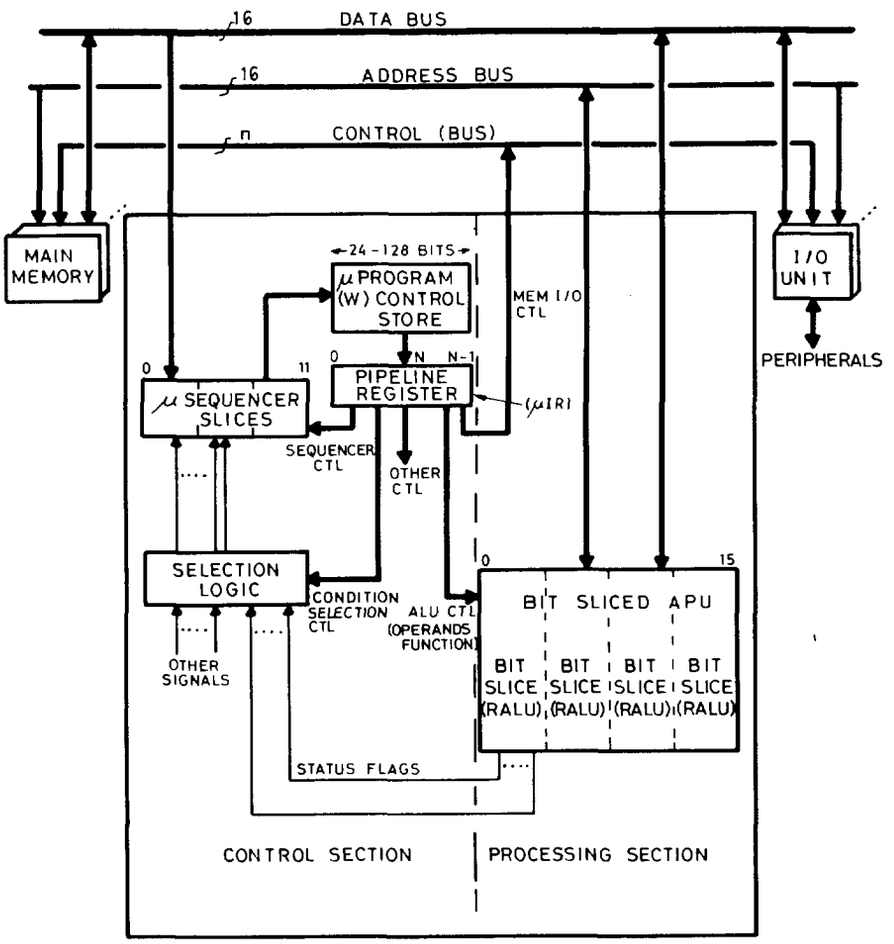


Fig. 6 Typical bit-sliced microprocessor design

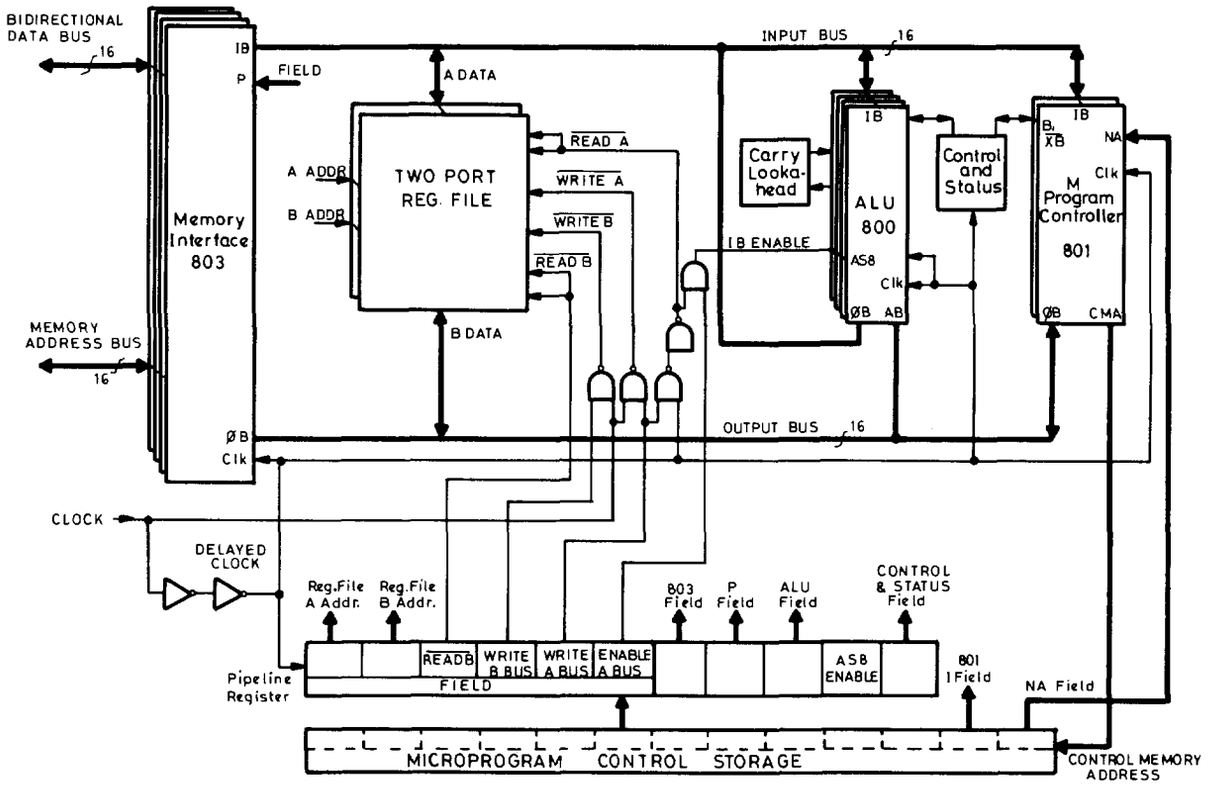


Fig. 7 Typical Motorola 10800 slice design

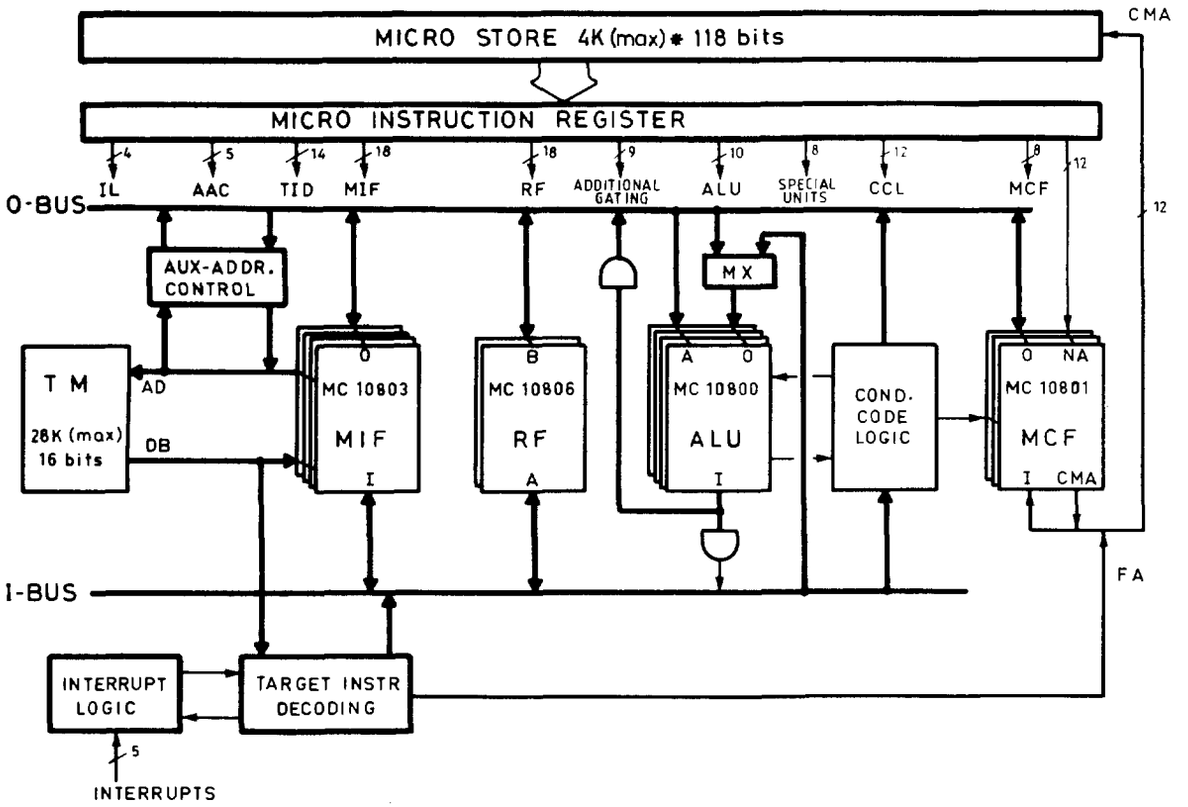


Fig. 8 MICE host block diagram for PDP 11E

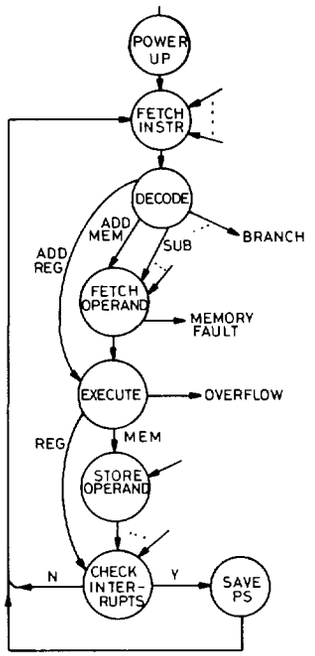


Fig. 9
State-transition graph for instruction execution

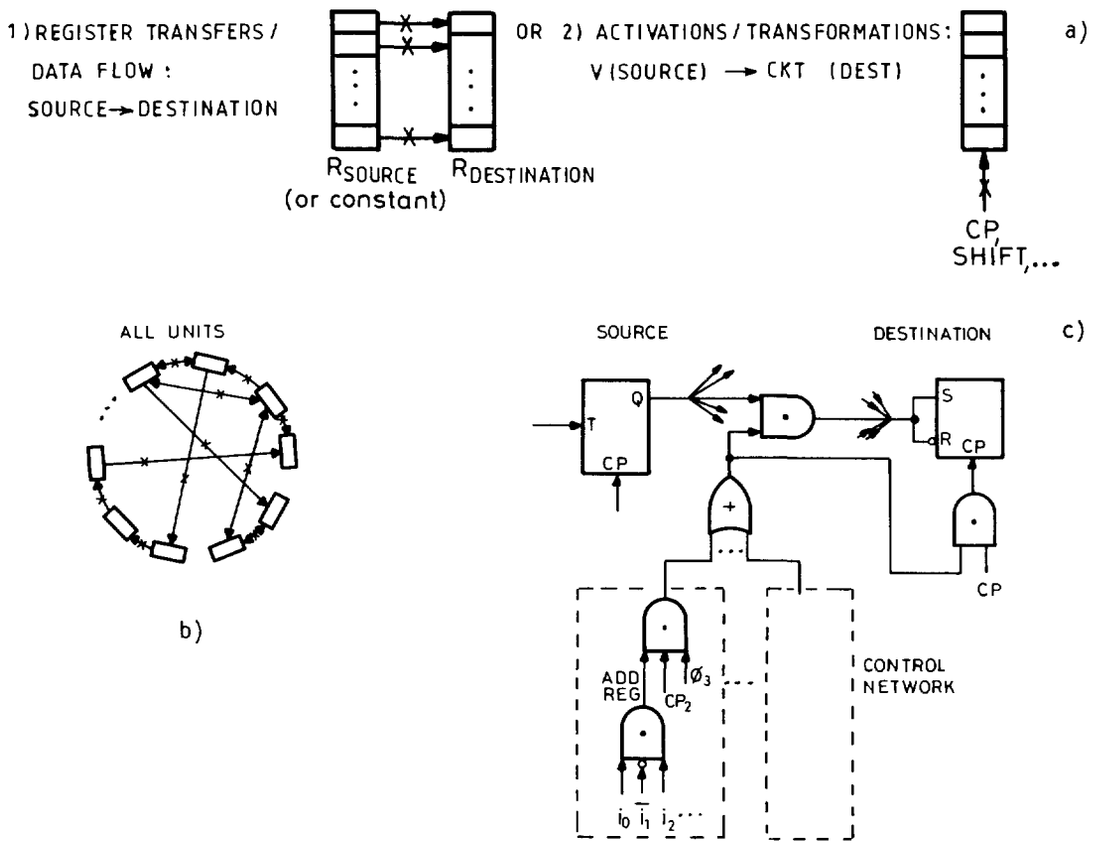
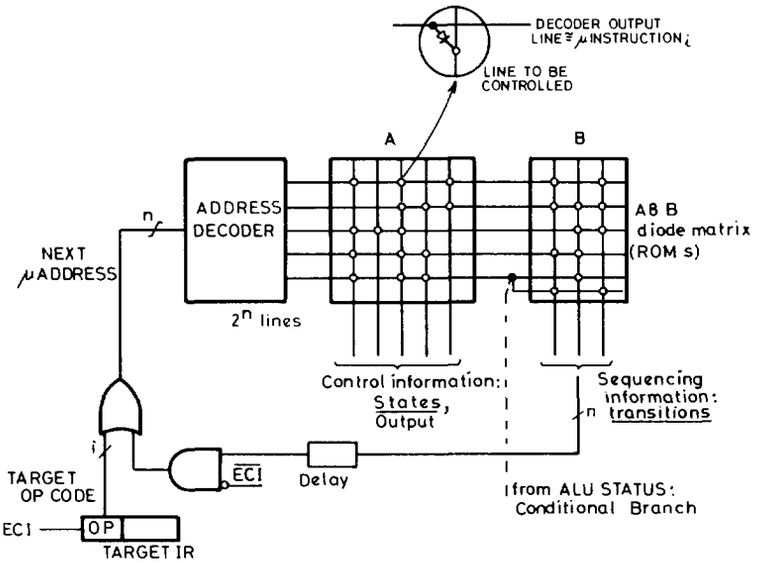


Fig. 10 (a) All primitive operations are either (1) or (2); (b) Itemize all registers, processing units, data and control paths; (c) Controlling a single source-information transfer.



Target OPCODE → μ instruction 1 + addr ($\mu instr_2$), $\overline{FF}_{ECI}$

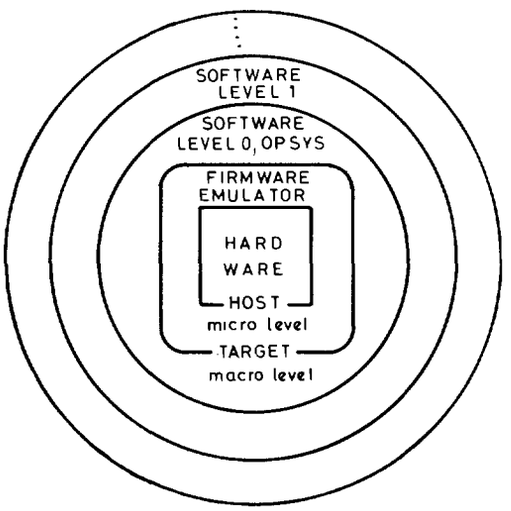
ADDR ($\mu instr_2$) → μ instruction 2 + addr ($\mu instr_3$), $\overline{FF}_{ECI}$

⋮

μ instruction N + FF_{ECI} (Flip-Flop stating end of current instruction)

Static target OPCODE → Dynamic sequence of host (μ) actions: each bit in each μ instruction controls a resource, action, data transfer, flag/status bit, etc.

Fig. 11 Wilkes microprogrammed control

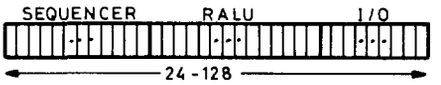


- Each layer creates an abstract/virtual machine for the level above, at 10:1 cycle cost.
- Emulator hides host idiosyncrasies, resources.
- Software layers selectively mask lower-level architecture.

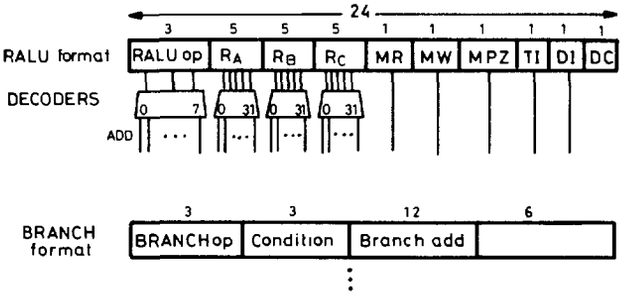
Fig. 12 Microprogramming terminology

(a) Horizontal, fully decoded (Wilkes model):

1 bit/resource controlled ⇒ fully parallel CTL



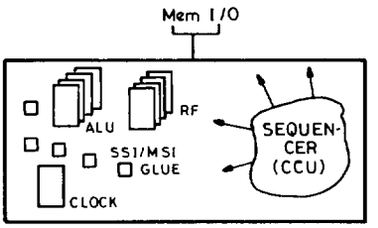
(b) Vertical, partially encoded, multi-format, ≃ "normal"



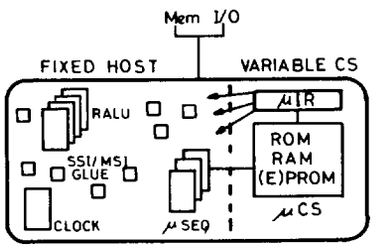
NB: Still messy, requires intimate hardware knowledge

- Horizontal hardest to "program", but potentially faster due to parallelism. A good μassmb with macros can approximate vertical.
- Vertical easier (e.g. "user microprogrammable" minis), but takes more instructions/cycles per macro operation since less potential parallelism.

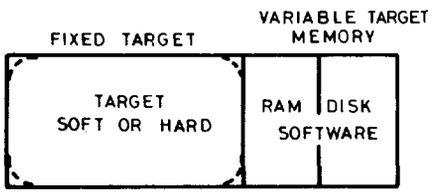
Fig. 13 Microprogramming distinctions



"Hard" wired target = fixed host.
(Fastest, but most complex, least regular.)



"Firm" μ prog'd target = fixed host + variable CS
(minis, main frames, even μP's: also programmable black boxes = SPEC PURP COMP).



"Soft" application system = fixed target +
variable TM easiest to change, most general,
but slowest.

Fig. 14 Hardware, firmware and software

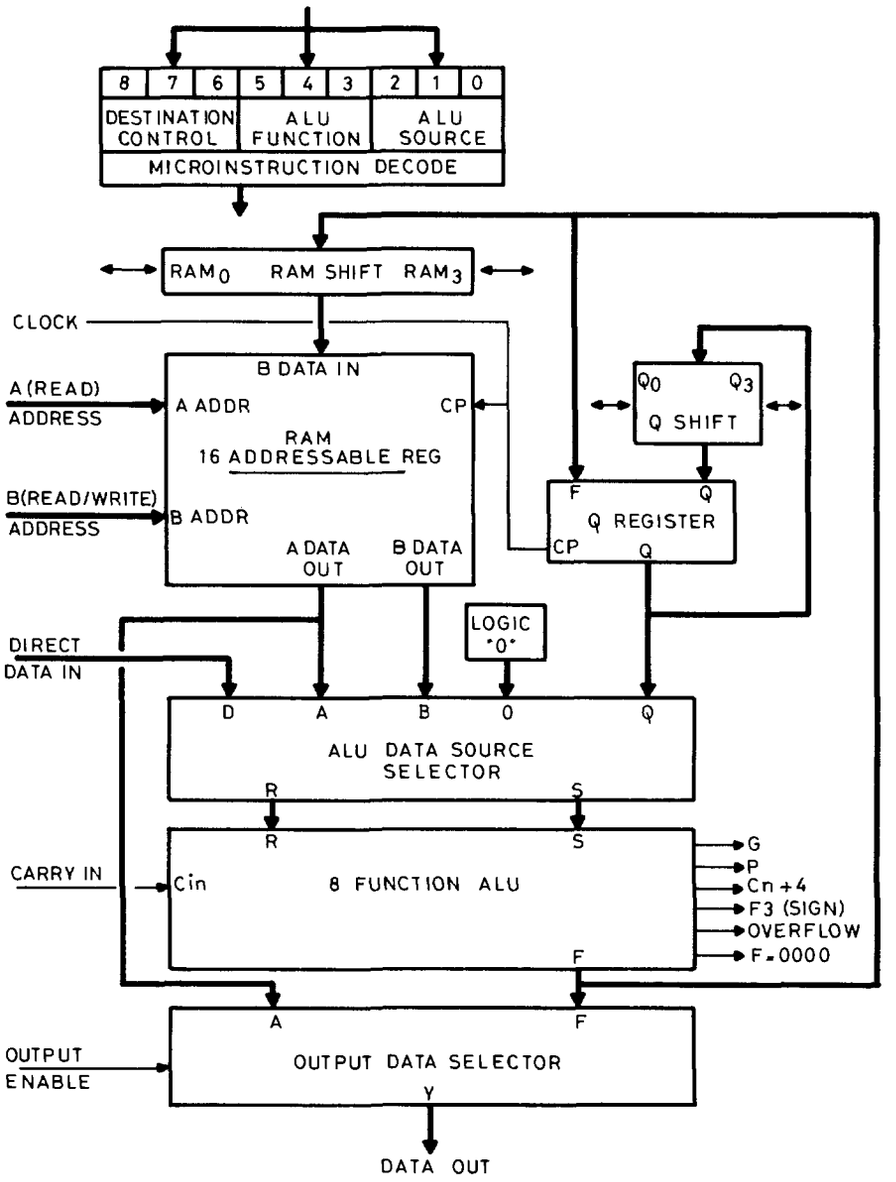


Fig. 15 4-bit AMD 2901 RALU slice

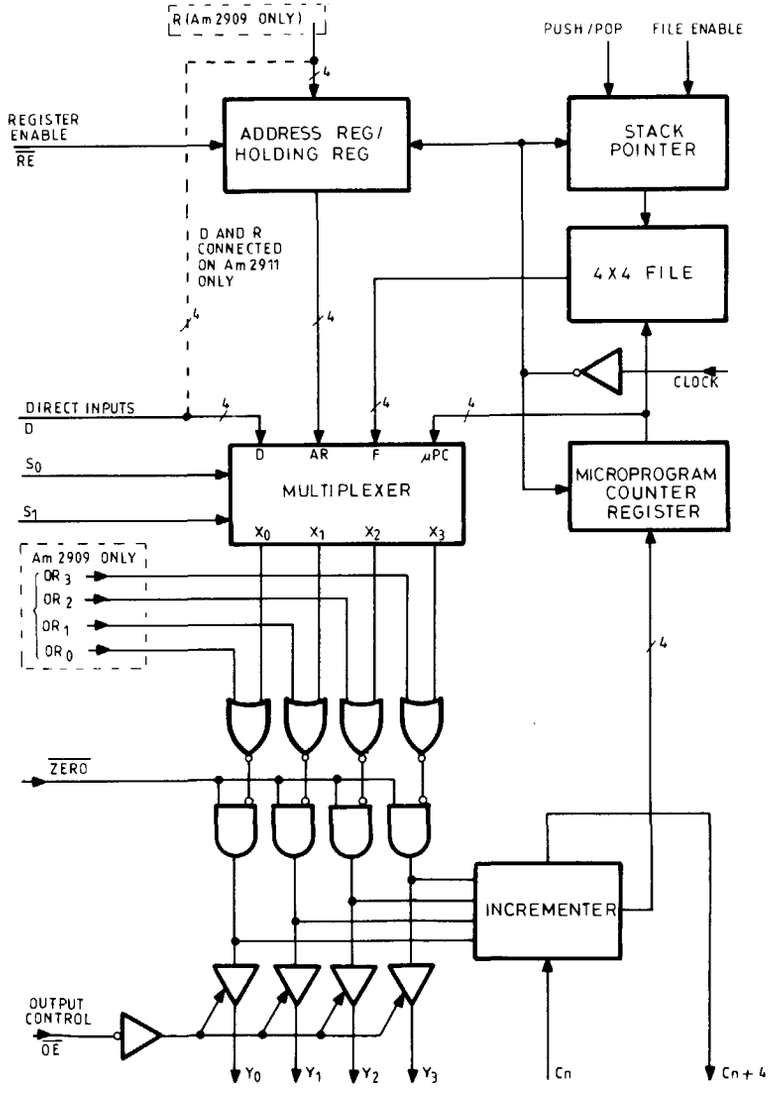


Fig. 16 AMD 2909 microprogram sequencer

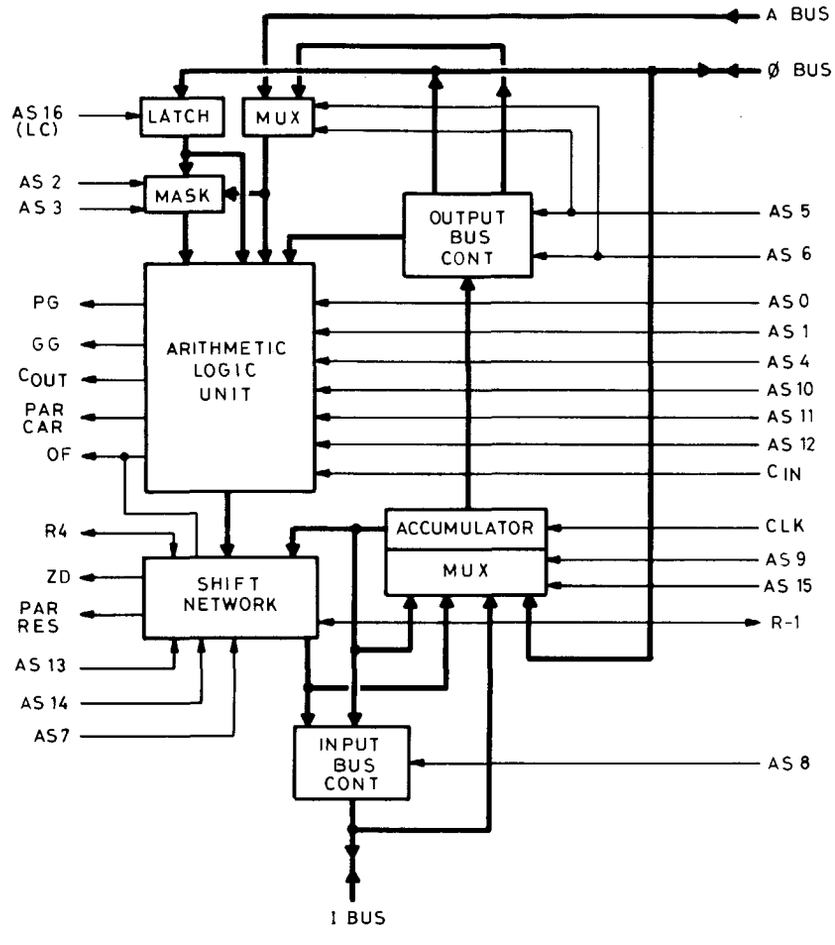


Fig. 17 4-bit ALU slice M10800

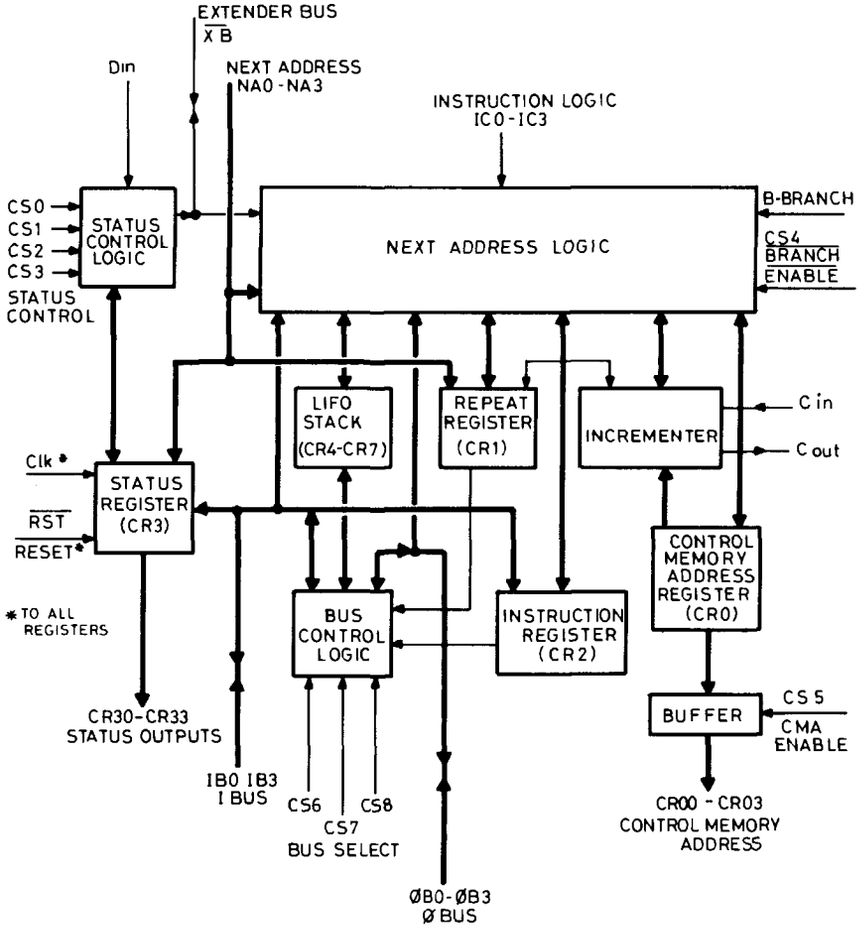


Fig. 18 4-bit microsequencer slice M10801

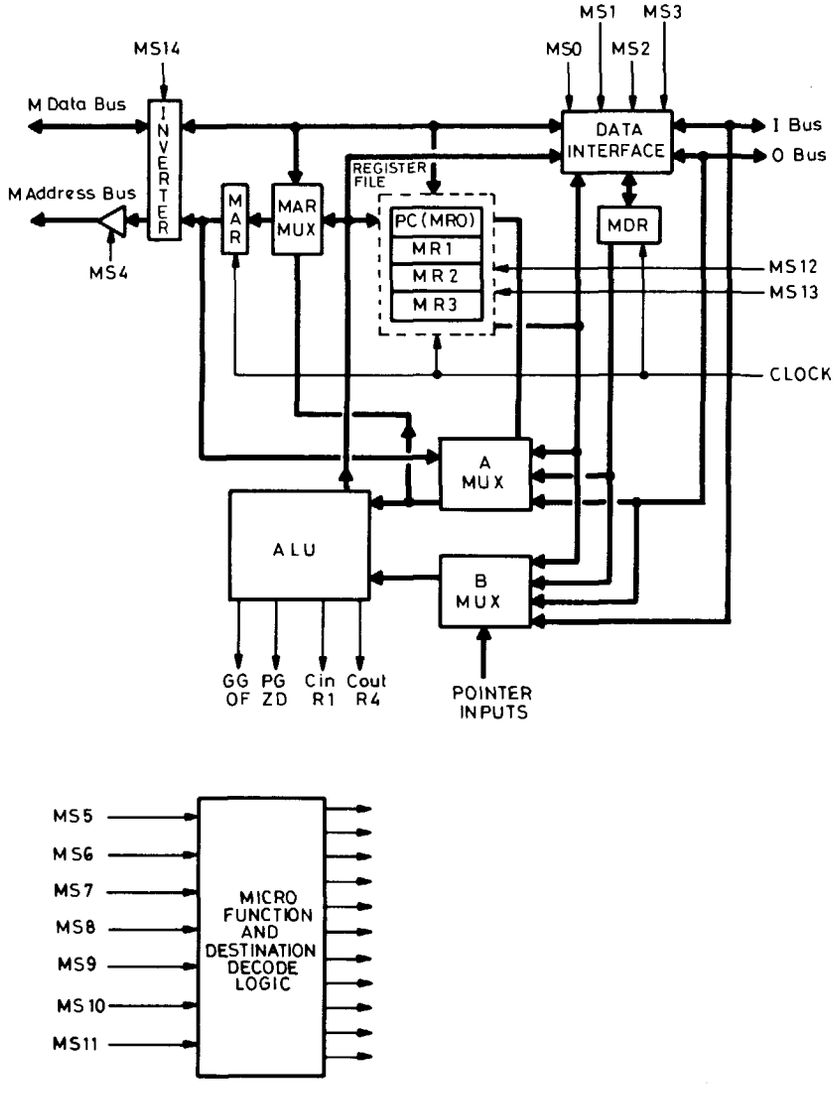


Fig. 19 4-bit memory interface slice M10803

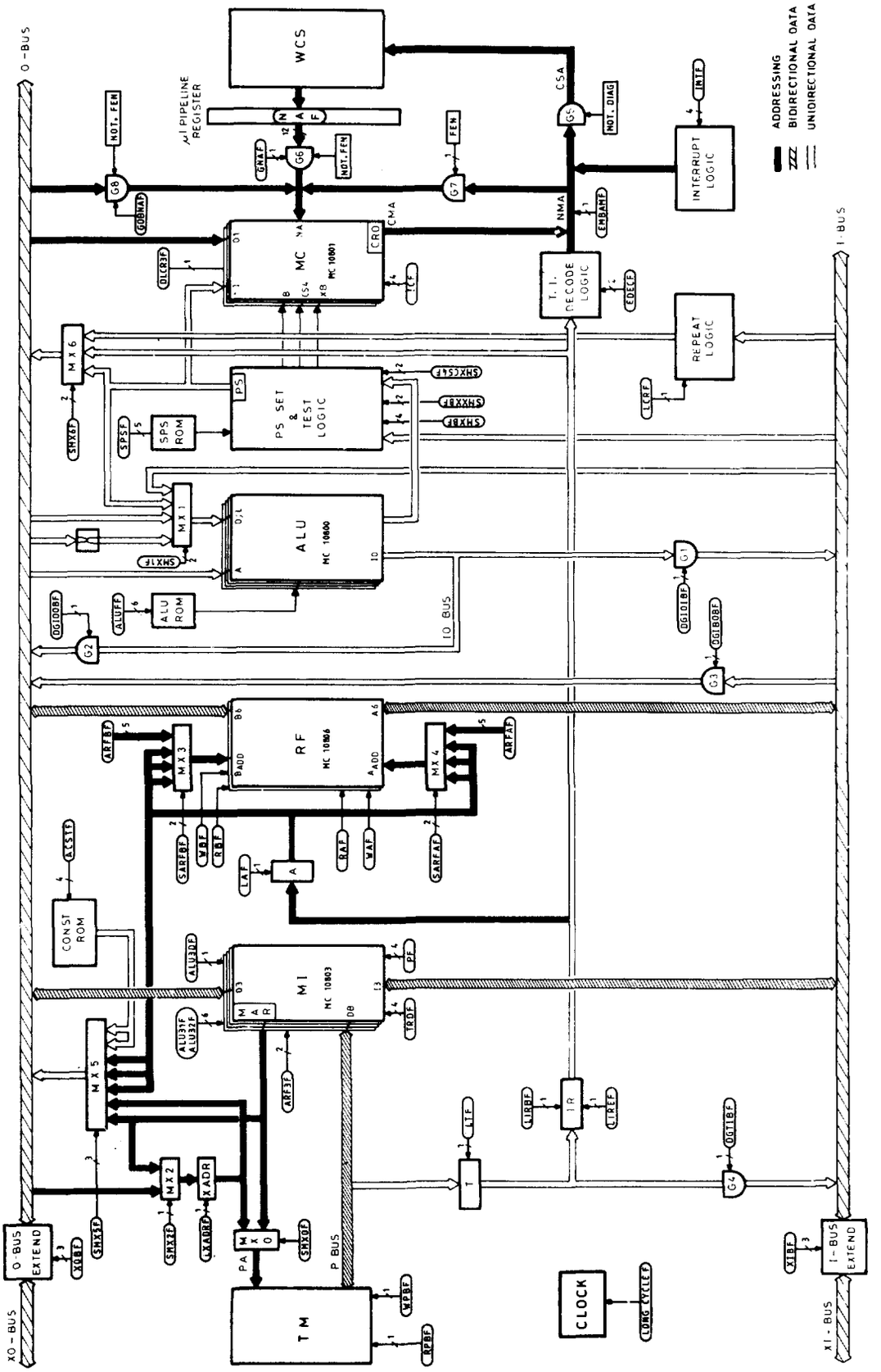


Fig. 20 Detailed MICE host block diagram for PDP 11E