The Use of PALS in CPU Design

by

David Winkel

Computer Science Department
Indiana University
Bloomington, Indiana 47405

TECHNICAL REPORT NO. 204

The Use of PALS in CPU Design

by

David Winkel

October, 1986

THE USE OF PALS IN CPU DESIGN

I) INTRODUCTION

As befits any healthy discipline, circuit design is a steadily evolving process. In the past we viewed synthesis in the intuitively obvious way: design with functional building blocks and map them directly onto silicon. Huge catalogs of MSI parts are a direct result of this view. For many years MSI was well matched to manufacturing know-how so the designer was in equilibrium with the tools and parts available to him. We are finally realizing that having everything you want at this level is too restrictive: we have to stock, test, integrate, and most of all understand far too many chips. We need to get away from chips into systems.

For systems work, VLSI is an obvious choice; you start with virgin silicon and craft a circuit with no restrictions other than those imposed by die size and routing constraints. This freedom is not without cost, however. Substantial financial, manpower, and CAD resources are required and limit its application to high volume parts where costs can be amortized. Recent programs (7) have made it available to the research community and potentially to small commercial projects, but it is still likely to be beyond reach of most practicing engineers.

Many desirable properties of VLSI can be provided by universal chips that can be tailored to specific applications, the so called ASIC (application specific integrated circuit). Gate arrays (8) are uncommitted chips produced in high volume at low cost and configured by wiring standard transistor unit cells into logic blocks at the final metalization stage. The costs are still non-trivial, but substantially less than full custom VLSI; turnaround times are also reduced to a few weeks. These factors have propelled the technology into an increasingly wide application area.

Standard cells are intermediate between gate arrays and full custom VLSI. The interface presented to a logic designer is similar to a gate array, but logic blocks are drawn from a library, then placed and routed on silicon by extensive CAD tools provided by the vendor.

What happens if you can't afford all this, but still need the power provided by ASIC technologies? Field programmable logic has long been a designer's dream and would be the ultimate ASIC technology if it could accommodate large systems. In the past it was available only for small state machines and replacement of random logic; as such it was the perfect micro ASIC technology. Newer PALs, however, equal the power of small gate array and standard cell devices, and will surely displace them for development and small production runs because of their low cost and short turnaround. As PALS become larger their application area will steadily encroach on gate arrays and standard cells if we can find appropriate techniques for managing their design.

Since many issues in using large PALs in system applications are complexity related we will, of necessity, have to adopt some techniques used by traditional ASIC designers to manage complexity. Depending on your viewpoint, this is a boon or distasteful since success comes from a higher level, top down, view of the design process with less emphasis on the art (tricks?) of optimizing microscopic chunks of logic. Unfortunately, some large PALs are poorly adapted to system design and we must find new ways of looking at them if we are to use them as ASIC devices.

We explore these issues by building a small CPU using a popular CMOS PAL with an unusually flexible logic structure (32VX10), using an ASIC design style wherever possible. The techniques we develop extend readily to more complex digital systems and future PALS of greater logic power. As far as we know, this is the first successful application of the 32VX10 in a system as complex as a CPU.

As we will see, the early stages of design are independent of logic type, but good PAL designs require very careful partitioning and assignment of logic to chips or the power residing in the fuse matrix will be wasted.

II) CHOICE OF TARGET ARCHITECTURE

We need a target CPU with these characteristics:

A) It must be large enough to reveal the real design issues in using large PALS.
B) It must be small enough to permit rapid construction.
C) It must have extensive diagnostics to verify the finished hardware.
D) It must be well described in the literature.
E) It is desirable for implementations to exist across a range of technologies so we can compare them to a PAL design.

Our choice is the DEC PDP8 which fulfills all of the above (6). We emphasize that the particular target architecture is of little importance: we are not interested in building a PDP8, but HOW to build one. The philosophy of design, algorithmic state machines, micro-coded control, and data path derivation are extensively treated in reference (1). The PDP8 is used in that book as a vehicle for showing the different techniques in action, and is a source for supplementary information to this technical report. Complete wire lists and micro-code are shown in (4).

We, in fact, have built this system in MSI, PALS, and are currently doing a 3 micron CMOS custom VLSI data path. We have done a paper gate array design, but have not implemented it for cost reasons. Control has been implemented with micro-code, hard wired state generators, and a PAL state machine. In each case, we require finished hardware to run the complete set of DEC diagnostics. Paper designs don't face this acid test, and thus may miss critical issues. This CPU is our local favorite for exploring the nuances of different technologies.

III) THE DESIGN PROCESS

So what parts of traditional design can we use? What changes? What ASIC design philosophies can we adapt? What special PAL properties are likely to impact our design process?

Design style is a curiously neglected area, perhaps because a traditional study of logic design emphasizes the microscopic transistor and gate aspects of the subject. Our experience leads us to start from the other end, working downward from system specification to ever smaller blocks until we reach something recognizably close to chips. The most important result from a top-down decomposition is that chips are chosen to fit a design instead of forcing a design to fit a preconceived chip set.

We think much like architects who start with an aesthetic building plan and let it dictate the shape and location of doors, windows,etc. This is not to deny that you can reverse the process and assemble predefined doors, windows, stairs, and other components into a house; it's just that

it will look like it. Elegant results require elegant tools.

Our foundation is hierarchical decomposition -- perhaps the most powerful tool yet found for managing complex engineering designs. The idea is essentially an application of the divide and conquer principle; the trick is to divide properly so the smaller units are easily conquered. If we insist that a decomposition yield a heirarchy of functionally related units this will usually be true. This is the guiding framework for top-down design and we will use it to advantage in this project. We anticipate that top level decompositions should proceed with little regard to implementation technology. The converse would be an indication of an insufficiently powerful formalism.

LEVEL 0 DECOMPOSITION

The standard textbook model of digital systems, figure 1a, is, of course, correct but gives little guidance to the designer.

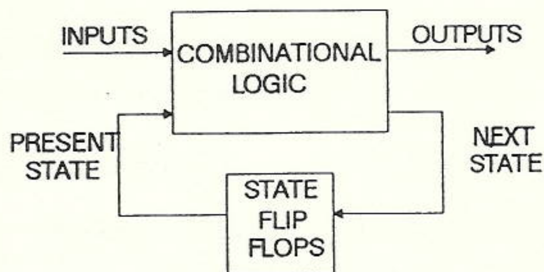

FIGURE 1 A CONVENTIONAL MODEL OF SEQUENTIAL SYSTEMS

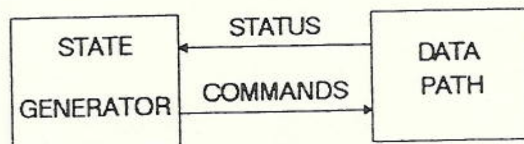

FIGURE 1 B FIRST DECOMPOSITION OF GENERAL DIGITAL SYSTEM

A decomposition into two cooperating entities, figure 1b, is always possible and desirable. This simple division is more profound than it looks, its possibility was shown in (2), its desirability arises from the orthogonality of the decomposition. Command and data-path are very different with well defined communication protocols between them. Because they are orthogonal each part can be designed with minimal consideration of the other. Instead of one large difficult problem we now have two much simpler, smaller, problems. It is the divide and conquer approach in action.

Control issues properly sequenced voltages (commands) to the data path, each pattern lasting for one clock cycle. The data path is an obedient slave to control and responds by carrying out some primitive micro-operation, often a transfer of data between registers, possibly modulated by logical or arithmetic operations. Control is an algorithm that choreographs these primitive micro-operations into a larger useful result. Examples might be: the fetch-execute cycle of a computer,or moving a head to a disk track. Useful algorithms nearly always have branching structures so they can respond to special events in the slave data path. The data path is responsible for generating proper status information and reporting it to control.

The independence of architecture and control permits exposition of each at

different levels. We use this to advantage here by discussing control at the micro-code level where algorithmic aspects are apparent. We will be able to suppress much detail in the control section and devote our attention to the PAL data path where the issues are not as standard. We usually do our initial algorithm development on a micro-code work station for convenience (5), even if later versions are to use state machines since the transformation to hardwired sequencers is straightforward. You should realize that although micro-code source looks like assembly language it is just a symbolic statement of voltage commands and their delivery sequence.

LEVEL 1 DECOMPOSITION

We decompose control and data path into smaller units. If done in true heirarchical fashion data path decomposition can proceed independently from control. While we know there must be some interaction, in practice it is surprisingly small.

We start by proposing abstract data path elements and connections we believe will support our target architecture. At this stage these are ideal elements with just the properties we want; it is a mistake to rush to data books at this stage since it prematurely moves us into a messy, detailed world. That will happen soon enough; it is better to put it off, move up a level,and design an elegant abstract machine.

Our hypothetical data path contains the following elements:

MA	a register which holds the memory address for the current read write memory location
MB	a register containing the data value to be written into memory
IR	the instruction register holds the current instruction during execution
PC	the program counter points to the next instruction location
AC	the accumulator holds temporary results and is always an implied operand to the arithmetic unit
LINK	a one bit register holding AC overflow
IE	a one bit interrupt enable register
INT	a one bit register that latches interrupt requests
SR	the control panel switch register
MEM	a 4k x 12 RAM
ALU	a general arithmetic logic unit
INPUT	the external input port
OUTPUT	from the AC

Many of these registers are specified in the PDP8 handbooks (AC LINK IE INT SR). Other hidden registers (MA MB IR PC) are common to all CPUs since they must have mechanisms to access memory, fetch and decode instructions, or maintain a pointer to the current control point. I/O, memory, and interrupt facilities are universal computer functions usually well described in programming manuals. These manuals also do a good job of showing possible transfers between user available registers. Thus, even for complex computers it is easy to postulate a complete set of data path structures.

No matter how these registers are divined, we must convince ourselves that data movement from source to destination registers (possibly modulated by arithmetic or logical operations) can emulate the PDP8 instruction set. The demonstration is constructive: write an algorithm to do it and you get control as a byproduct. For example, instruction fetch must involve these micro-operations regardless of final chip details:

PC--> MA; start mem read cycle; wait for mem done; MEM-->IR

Now we are in a position to design the control algorithm to supervise data path activity. We can go very far without knowing exact details of the data path chips, a reflection of the orthogonality of the two parts. For example, a control command to move data between two registers really doesn't have to know at this stage whether they are JK or D registers, although that will arise in later stages. The algorithm can be specified by state diagrams, micro-code, or ASM charts. ASM is our preferred formalism for hardwired state machines since it supports a combination of Mealy-Moore outputs (1). Whatever formalism we use there are standard cookbook procedures for converting a symbolic algorithm to hardware so we can safely stay in the symbolic realm. In the next level we will be led safely to chips.

The complete algorithm is shown in reference (4), relevant portions are shown in Appendix A where we assume an industry standard micro-sequencer (AMD 2901). This algorithm requires the register operations in Table 1. We suggest that you manually trace the first 4 states which fetch the next instruction and load it into the IR. For simple CPUs manual tracing of the algorithm - data path interaction is sufficient to demonstrate conformance to the target architecture. However, we prefer a register level simulation running against standard diagnostic programs in simulated memory for a more rigorous demonstration. This is essential for gate arrays where it is costly to turn around new versions. With the advent of reprogrammable PALs simulation is less important; it may be easier to build the hardware than the simulator. There are several register transfer simulators available, but we use a simple one locally written in a variant of LISP. In any event, a register simulation is a fairly coarse tool that will reveal major faults, but miss subtle timing issues.

LEVEL 2 DECOMPOSITION

Here is the first place we meet chips. We have an excellent description of their desired properties from the-top down decomposition. Can we find a good fit with available hardware? With MSI this is answered by a simple tour through data books. With VLSI you have the freedom to design a perfect match; with PALs it may not be so easy. Let's look at some PAL characteristics that affect design:

1) PALs are strongly pin limited.
   We may not have enough pins to accommodate the target logic blocks.
2) PALs have a fixed AND-OR structure.
   A common problem is restricted width of the OR structure
3) Most PALs use D type flip flops.
   These are poor matches to register protocols. JK or enabled D is desired.
4) There is excess logic power in the fuse matrix. We must find innovative ways to use it to advantage.

HOW TO LIVE WITH A PAL.

A) PARTITION CLEVERLY

Partitioning is a standard ASIC problem, and we can use the same techniques with PAL design. While it lies outside the logic realm it impacts logic indirectly by dictating how much of it we can package per chip. The basic idea is simple: partition a circuit diagram into blocks that contain lots of logic with few wires between blocks.

B) EXPLORE SUBTLE PROPERTIES OF THE AND-OR-FLIP FLOP PATH
Some PAL structures allow us to modify internal type D storage into the desired JK or enabled D protocol. They permit the standard register paradigm.

C) USE THE FUSE MATRIX CREATIVELY
There is excess logic power here that can be used in novel ways. We can set up parallel data paths to each register, each with a private logic unit. We can move parts of command and status generation into the fuse matrix.

We will have to exploit all of these, let's explore each in turn. Finding a good partition is critical in this project. Fortunately, a standard CPU partition has long been known, namely bit slices. This is based on the observation that simple logical commands operate on bits independently. For example, in a complement AC operation bit 5 is inverted independently of bits 4 or 6. Whatever circuitry is involved is duplicated in each bit position and has no communication with its neighbors. When we include memory and an ALU we arrive at the standard bit slice partition shown in Figure 2. For a 12-bit machine like the PDP8 There are 12 identical slices working in parallel for all simple register transfers and logical operations. Each slice has one bit of the MA, MB, IR, PC, and AC registers with internal operations controlled by shared command lines.

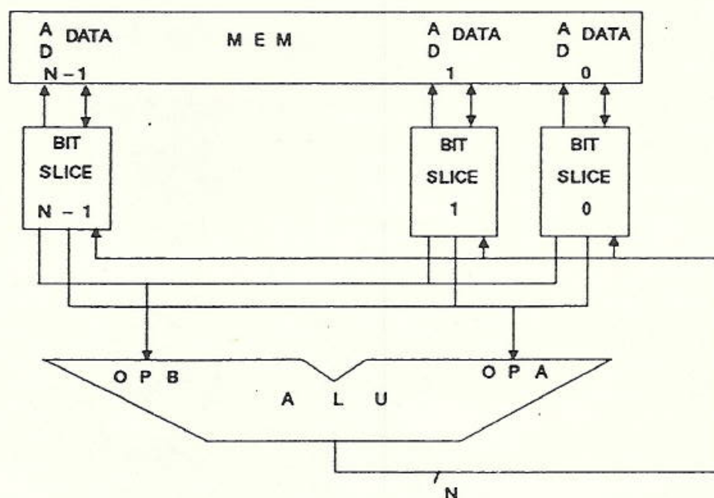

FIGURE 2 CONVENTIONAL BIT SLICE ORGANIZATION

Now comes the big if -- IF we can fit an entire bit slice into a single PAL we have an excellent chance of doing an elegant design. If not, consider abandoning the project since further partitioning almost always yields blocks with increased communication requirements. There are no hard and fast rules and little accessible theory to guide you to good partitions. Fortunately intuition is pretty good in this area - let common sense be your guide.

The bit slice model does have interslice communication for shift and arithmetic commands. Right and left shifts are easily handled by allocating two bidirectional pins per slice -- one for input, one for output -- and connecting adjacent slices with a wire carrying shift-in/ shift-out data. This is straight forward, but uses up two of our precious pins. There is another possibility that will be clear after we discuss arithmetic units so we hold off on this decision for now.

Carry propagation during arithmetic operations is a hidden interslice communication problem that we must allow for. We have two options:

1) Use an external arithmetic unit.
   We are moving carry propagation off slice to a word wide chip set optimized for fast carry propagation. In this approach we need to dedicate two pins to feed the ALU and one to receive its result. This is the fastest way to do add/subtract which are inherently word wide operations. Many external ALUs are capable of shifting so we can share the ALU pins for shift operations at no extra pin cost.

2) Put the arithmetic operations in the fuse matrix.
   This is easy and we also pay no pin penalty since we can share two shift-in/out pins with carry-in/carry-out. The advantage is reduced chip count by eliminating the external ALU; the disadvantage is slower arithmetic operations because the carry must ripple propagate.

Since we are looking for a minimum package count in this exercise we choose the second alternative. Also, it uses up a slight amount of the excess logic in the fuse matrix.

So we now have a picture of the pins required per bit slice derived strictly from partitioning arguments:

registers:	MA, MB, IR, AC, PC, LINK
carry/shift path:	RO/LI, LO/RI
PDP8 specific items:	MEM(4k x 12), panel switch register (SWR), INPUT
control commands:	C0–C4, DIR
pattern port:	discussed later under status generation.

It is shown in Figure 3.

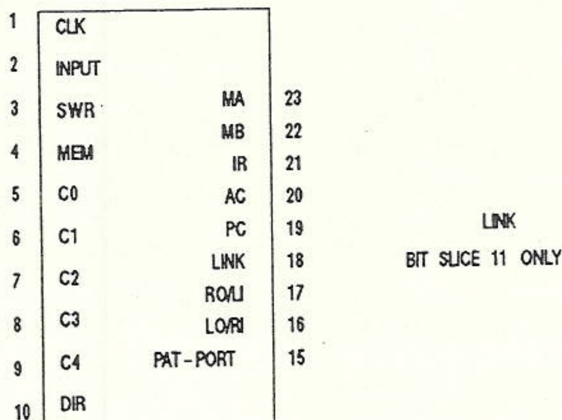

FIGURE 3 PIN ASSIGNMENTS ON 32VX10

The five command pins suffice to set up 32 different register transfer micro-operations which are adequate to select among the 30 entries of Table 1. The special DIR control pin sets the direction for the tri-state buffer on the carry/left-shift path. In principle, this could be derived from the 5 command bits, but there is inadequate AND-OR logic on that path so it must be externally decoded.

Now for the go-nogo decision. Can we fit the logic of Table 2 into the PAL of Figure 3? The answer in not an immediate "yes" but by being clever with the XOR structure of the 32VX10 we can do it. The problem is to make the type D flip-flop implement a register transfer protocol. A register must maintain its contents for many clock cycles until selected by control as a destination at which time it must load in new data. But, the D flip-flop loads every clock cycle. The desired register exitation function is:

```
eq 1                                        L     = load
          Qn+1 = /L * Qn  +  L * ND        ND    = new data
                                            Qn    = flip-flop Q after n clock
                                                    pulses
                                            Qn+1 = Q after one  more clock
                                                    pulse
```

A simple attempt at generating Table 1 operations with the standard AND-OR PAL fuse matrix shows the inadequacy of that structure. The register protocol of eq 1 is not only desirable, it is necessary.

Figure 4 shows the way an enabled D is synthesized in conventional MSI. This clearly shows the recirculation path when LOAD is false and the insensitivity to noise on the load line except at the active clock edge. The ability to ignore glitches is of the utmost importance when signals are generated inside a chip since we have no way to probe internal nodes. We must be correct by design. Figure 5 shows how an XOR in the flip flop path allows us to realize the desired protocol. The relevant cases are:

a) recirculate when L = 0 (L=LOAD ND=NEW-DATA Q= PRESENT FF VALUE)

```
     Y = L * (ND XOR Q)                    Z = Q XOR Y
                                             = Q * 0 * (ND XOR Q)
                                             = Q XOR 0
                                             = Q
```

b) load new data when L = 1

```
     Y = L * (ND XOR Q)                    Z = Q XOR Y
       = 1 * (ND XOR Q)                      = Q XOR (Q XOR ND)
       = (ND XOR Q)                          = (Q XOR Q) XOR ND
                                             =     0     XOR ND
                                             = ND
```

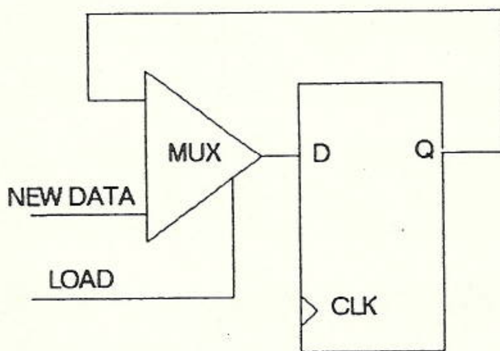

FIGURE 4 ENADLED D FF

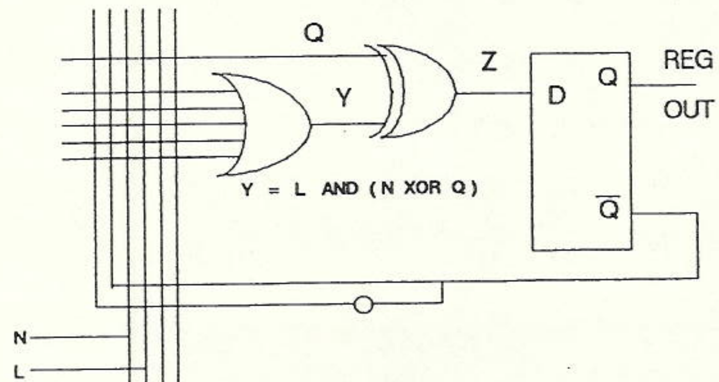

FIGURE 5 CHANGING TYPE D FF TO ENABLED D TO SUPPORT REGISTER PROTOCOL

As (3) shows the AND-OR-XOR structure is capable of much more. We can use it to convert the type D flip-flop to any other kind, a JK for instance. A really striking result is that we can generate ANY logic function of inputs to the fuse matrix. Each register now has its own private logic unit, and they operate in parallel! This gives a new dimension to PAL bit slices, as compared to conventional ones with just a single internal data

path. Later we use this result to cut states from our control algorithm, thereby speeding up our system compared to standard bit slice architectures. It is a pity the XOR structure is not standard in PALs. It appears to be hidden in some vendor's storage cells, but not in a form that can be modified on the fly, as in the 32VX10.

We have come a long way to a final hardware realization of our top down CPU decomposition. In fact, we have done more than create a data path. Refer to the expanded level 0 decomposition, Figure 6A, of our standard MSI benchmark implementation. The controller is broken down into a pure state generator and logic which converts state and status into control signals. In similar fashion, the data path is decomposed into pure register transfer elements and status generation logic. In each block the number of MSI chips devoted to it is given in parenthesis. Note the complexity of command and status generation. It is tempting to move this into the left over logic of the bit slices.

Refer to the 44 chips used for command generation. Their sole function is to convert state into 34 control lines for the data path. Why do we have so many command lines? In MSI each register requires separate control: the ALU requires 5 bits to specify what it does, 3 bits are required to specify a source register for the ALU. It goes on and on until suddenly you have far more required control information than you thought necessary. The reason there are so many is each data path element requires its own control field, and there are lots of separate elements. MSI forces small partitions to match chips and the explosion of wires between chips is a reflection of a non-optimum partition.

Consider the PAL bit slice which contains just as many registers as the MSI version. Pin limitations have forced us to send in a 5-bit command field which is INTERNALLY decoded into multiple data path control. Without thinking about it we have moved a huge chunk of command generation logic inside the PAL. Interesting! Thus 40 of the 44 command generation chips and 34 data path chips are now hidden inside the 12-bit slice chips. We have replaced 78 chips by 12, Figure 6b. Now there's power for you.

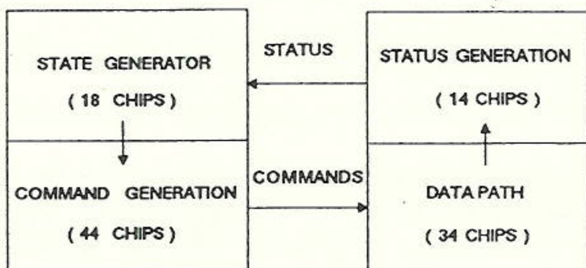

FIGURE 6A

CHIP COUNTS FOR MSI IMPLEMENTATION

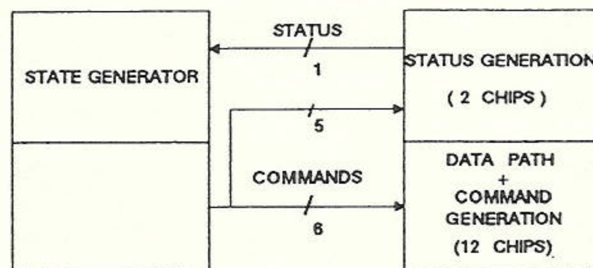

FIGURE 6 B

CHIP COUNTS FOR PAL IMPLEMENTATION

Can we be as clever with status generation logic? Status is often concerned with the presence of patterns across a multi-bit register, but bit slices don't know about their nearest neighbors, so this information must be detected outside the slices. We are forced to use a pattern checker oriented perpendicular to the bit slices. A list of the patterns that must be detected is shown in Table (3). Once again we are presented with a decomposition and seek an elegant representation in real chips. The abstract pattern checker is shown in the lower part of Figure 7. Its function is to check a 12 bit field for the patterns shown in Table (3). Pattern type can be specified by only 5 bits and the matcher reports its

presence or absence back to control to affect possible algorithm branching.

These elements cooperate in the following way. The state generator knows that a succeeding state will test some register for a certain pattern. In preparation, it latches the register of interest into the pattern register where it will be stable during the following test state. Upon transition to the test state the algorithm generates a pattern number and sends it to the match PAL and waits for a single bit status reply (equal or not equal) to set up a two-way branch at the end of the test state.

Again we are left with lots of unused logic, but there is no way to use it for other functions. Indeed, there is not much unimplemented logic left. The best we can do is throw in a few one-bit status flip-flops like, HALT, IE, and INT. The remaining status information concerns the front panel switches which are tested in a switch-test PAL for report back to the algorithmic state machine.

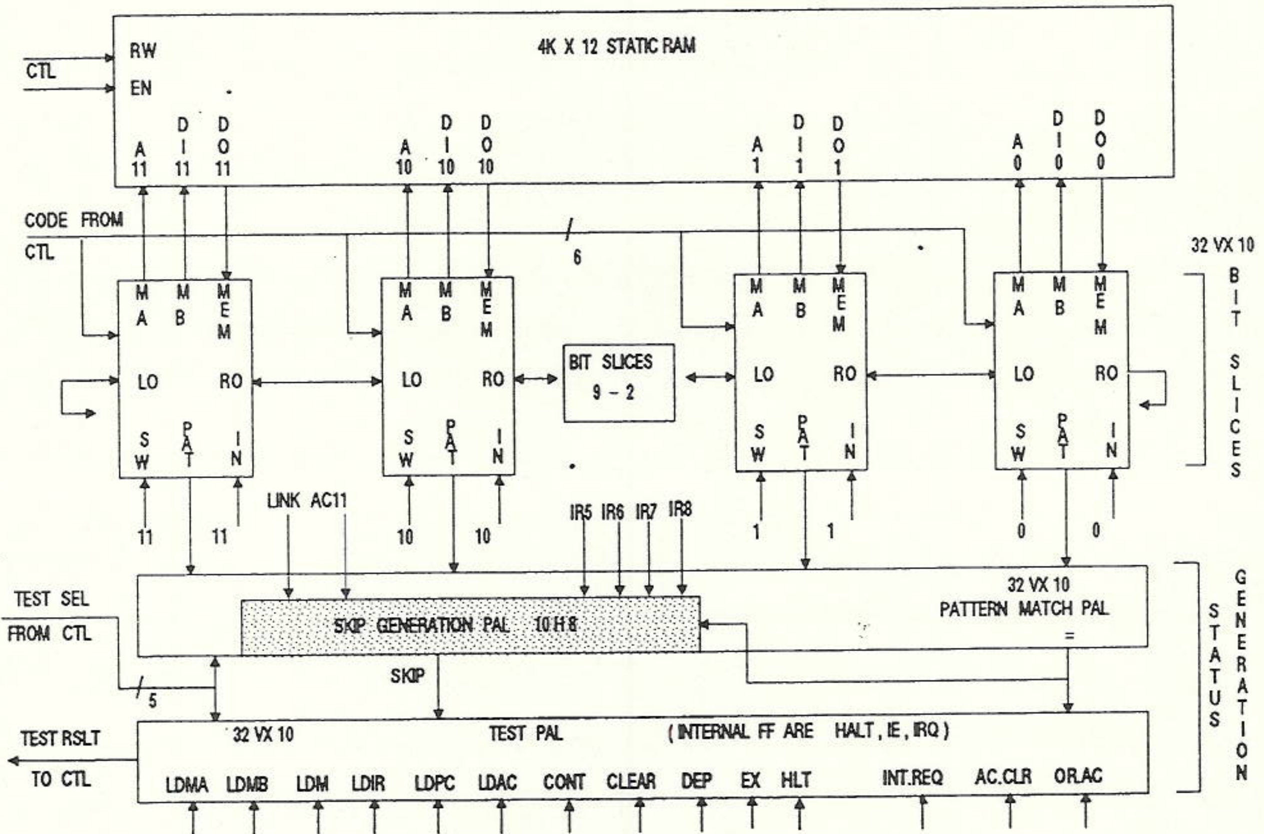

FIGURE 7 PAL IMPLEMENTATION OF PDP8

The final results are impressive: 92 MSI chips replaced by 14 medium and and one small scale PALs, Figure 7. The fuse patterns are implied by the equations shown in table 2. We have not discussed control since there are well known techniques for implementing state machines. Our testing was done on a micro-code work station for convenience (5).

Figure 7 is very close to a final wiring diagram and is remarkable for its paucity of wires. Where did they all go? A byproduct of PAL bit slices is the use of the fuse matrix as a general interconnection device, in addition to its logical properties. Viewed in this fashion we have a crossbar switch with 100 switch points, something no designer would dream of building because of it's expense, but we get it free. A remarkable luxury.

A COMPARISON OF PALS TO OTHER TECHNOLOGIES

1) MSI

Our MSI version has 92 chips and 1290 wires. There are not many reasons for doing it this way, although there are some niche applications where it is still appropriate. Very small systems may fit MSI well. Some applications do not map well into PALs, and you may have no other choice. This will surely become less important as more powerful PAL devices are produced. Some applications have highly optimized MSI chip representations which will be replaced with difficulty; arithmetic units come to mind.

2) CONVENTIONAL BIT SLICES

Standard commercial products are 4 or 8 bit slices through an ordinary register CPU. If your target architecture is a standard register machine bit slice chips will probably host it without much forcing. In this case it comes close to being the technology of choice. There are minor problems of implementation: commercial bit slices require many command bits (24 in the 2903). Memory requires a dedicated MA register, which means you need a separate hardware register copying one of the internal bit slice registers. These are still minor quibbles. Major problems arise when your target architecture is non-standard, a stack machine for example. Then your options are limited to lengthy micro-instruction sequences to emulate stack behaviour with the bit-slice internal register set.

3) VLSI

This is a pleasant method if you have extensive design aids, can tolerate sizable development costs, live with fabrication delays, and enjoy working at the primitive silicon level. Working directly in silicon does have a special set of constraints, however. At first sight you can tailor data-paths and control in arbitrary fashion. This is not true. Wires, not logic, are the constraint in VLSI. Many things that can easily be done at the one or two-bit level break down for replicated data-path structures, so you choose simple logic that lays out well on a two dimensional surface. Pin constraints are minimal with pin grid packages so partitioning becomes less of an external problem. The cost of one design/fabrication cycle is so high that extensive simulations of functionality and timing become a part of the design process throughout all stages.

4) GATE ARRAYS

Here we have a wide choice of design specification. Many vendors support a master document in MSI logic format using some subset of standard TTL chips. The vendor partitions this circuit, reduces the MSI specification to a logical gate array equivalent, auto-routes the metal path, tests and packages the dice, and delivers the finished product. Naturally these services are not free, but there are many applications where this technology is cost effective. At the other end of the spectrum, the user can wire up the primitive gates. In any event, the user should not abdicate total responsibility for there are testability and partitioning issues that must be incorporated from the beginning. One of the real pluses of gate arrays is the ability to fit an entire system in one package, thereby eliminating the partitioning problem altogether. Package pins range from 16 to over 200, which provides a good ratio of pins/logic. A PDP8 can easily be packaged in a one-chip gate array.

5) PALs

At present this is our favorite technology for special CPU/memory architectures for several reasons:

a) Field programming is an excellent development tool for rapid prototyping.
b) Field reprogrammability is an even better tool.
c) It is by far the cheapest way to develop systems.
d) Having multiple internal parallel data paths, each with a private logic unit, is unique to PALs. This leads to maximum speed implementations.
e) Fuse patterns can be automatically derived in certain cases which gives correctness by design. This is relatively easy with PALs, much harder in other technologies.
f) We are not limited to standard register/von Neuman architectures. Stack machines or special list memory architectures are readily mapped into PALs, but are difficult in other technologies.
g) Compact systems with few wires are natural results.

REFERENCES

1) Prosser, Franklin P., and Winkel, David E., "The Art of Digital Design", 2nd ed. Prentice-Hall, Englewood Cliffs, N.J.,1986

2) Snyers, D., and Thayse, A., "Algorithmic State Machine Design and Automatic Theorem Proving: Two Dual Approaches to the Same Activity", IEEE Trans. Comput., vol.C-35, pp 853, Oct., 1986

3) Winkel, David E., Tech Report 188, IU Comp. Science dept., Bloomington, Ind.

4) Prosser, Franklin P., CS421 Lab. Notes, IU Bookstore, Bloomington, Ind.

5) Prosser, Franklin P., and Winkel, David E., "The Logic Engine development System", Proceedings of MICRO-16, October 1983, pp84-91.

6) Bell, C. Gordon,"Computer Structures: Readings and Examples", Ch 5, Mc Graw Hill, 1971.

7) MOSIS Project, USC Information Sciences Institute, 4676 Admiralty Way, Marina Del Rey, CA 90292-6695.

8) "CMOS Macrocell Manual", LSI Logic Corporation, 1551 McCarthy Blvd., Milpitas, CA 95035.

APPENDIX A

MICRO-CODE SOURCE FOR A PDP8

*/ The complete micro-code, micro-assembler conventions, and micro-code development station are described in (1). We have relegated some of the less interesting sections like manual switch processing to that reference; the register transfers listed in table 1 are derived from the complete control algorithm however.

Complete wire lists are given in (4) /*

*/ This algorithm is shorter than reference (1) because we have parallel data paths and logic units. Also, the test PAL has enough intelligence to collapse

interrupt test states. A ;*** in the comment field shows where more powerful PAL data-paths have shortened (speeded up) the algorithm. /*

```
FETCH        EQU     *
             JUMP   IDLE IF HALTFF=%T JUMP  F2   IF NOINTRPT=%T          ;***
PROCESS.INT JUMP    EXECUTE ;JAM.JMS
F2           CALL   READ.TO.IR
             JUMP   EXECUTE IF NO.MEMORY; EA.TO.MA
F3           CALL   READ.TO.MB
             JUMP   EXECUTE IF DIRECT.ADDRESSING
F4           CALL   AUTO IF AUTO.INDEXING
F5           JUMP   EXECUTE IF NO.INDIRECT.OPERAND; MB.TO.MA
F6           CALL   READ.TO.MB

EXECUTE      JMAP   ;;instruction decoding in mapping ROM
AND.CODE     JUMP   FETCH; MB.AND.AC.TO.AC
TAD.CODE     JUMP   FETCH; MB.PLUS.AC.TO.AC  COND.COMP.LINK   ;***
ISZ.CODE     CALL   WRITE; INC.MB
             JUMP   FETCH; COND.INC.PC                        ;***
JMS.CODE     CALL   WRITE; PC.TO.MB
             JUMP   FETCH; INC.MA.TO.PC
JMP.CODE     JMP    FETCH; MA.TO.PC
IOT.CODE     */ see reference (  )
OP.CODE         "       "          /*
```

TABLE 1 DATA PATH OPERATIONS

CODE	SYMBOLIC NAME	OPERATION	PAT. PORT DFLT=IR	DIR	LINK DFLT=LINK
0	HOLD	PRESERVE ALL REGISTERS	PAT.PRT	Z	DFLT
1	SW.TO.MA	SWITCH REG --> MA	DFLT	Z	"
2	SW.TO.MB	" MB	"	Z	"
3	SW.TO.PC	" PC	"	Z	"
4	SW.TO.IR	" IR	"	Z	"
5	SW.TO.AC	" AC	"	Z	"
6	PC.TO.MA.MB.PCINC	PC --> MA,MB PC+1	"	LFT	"
7	MEM.TO.IR	MEM(MA) --> IR	"	Z	"
8	PG0.TO.MA	%(00000)(IR6-IR0) --> MA	"	Z	"
9	CURPG.TO.MA	%(MB11-MB5)(IR6-IR0) --> MA	"	Z	"
10	MEM.TO.MB	MEM(MA) --> MB	MA	Z	"
11	INC.MB	MB+1	MB	LFT	"
12	JAM.JMS	%100000000000 --> IR	DFLT	Z	"
13	CLR.LINK	CLEAR LINK	"	Z	0
14	AND	MB AND AC --> AC	"	Z	DFLT
15	PLUS	MB plus AC --> AC			
		complement LINK if ovflo	"	LFT	cond.com
16	AC.TO.MB	AC --> MB	"	Z	DFLT
17	PC.TO.MB	PC --> MB	"	Z	"
18	INC.MA	MA + 1	"	LFT	"
19	MA.TO.PC	MA --> PC	"	Z	"
20	INC.MA.TO.PC	MA + 1 --> PC	"	LFT	"
21	CLR.AC	0 --> AC	"	Z	"
22	INPUT	INPUT OR AC --> AC	"	Z	"
23	COM.LINK	complement LINK	"	Z	comp
24	COM.AC	complement AC	"	Z	DFLT
25	INC.AC	AC +1 conditionally comp LINK	"	LFT	"

```
26   RT.SHIFT         AC,LINK rotate rt       "    RT    AC11
27   LFT.SHIFT        AC,LINK rotate lft      "    LFT   AC0
28   AC.OR,SW         AC OR SWITCH REG        "    Z     DFLT
29   SKIP.AC=0        PC + 1                  AC   LFT   "
```

Table 2 SELECTED FUSE EQUATIONS FROM TABLE 1

```
code c4-c0     AND - OR equations                     comments

2    00010     /c4*/c3*/c2*c1*/c0 * MB * /SR  +       load MB from SW:
               /C4*/C3*/C2*C1*/C0 */MB * SR           MB = LD*(SW XOR MB)

18   10010     C4*/C3*/C2*C1*/C0  * RI                increment MA:
                                                      RI = carry in
                                                      sum = MA XOR carry in
                                                      MA is one input to XOR
               LO = /c4*/c3*/c2*c1*/c0 * MA * RI      carry out = MA * carry in

25   11001     c4*c3*/c2*/c1*c0 * RI                  for least sig 11 bits
                                                      increment AC is like inc
               LO = c4*c3*/c2*/c1*c0 * AC *RI         MA

               c4*c3*/c2*/c1*c0 * RI                  in most sig bit ovflo must
                                                      complement LINK
               LINK = c4*c3*/c2*/c1* c0 * AC *RI      LINK is input to XOR
```

TABLE 3 STATUS GENERATION

```
test

0    NOINTRPT              = (IR eq 110000000001) + /IE + /IREQ
1    NO.MEMORY             = (IR eq 11----------) +
                             (IR eq 1--0--------) +
                             (IR eq -110--------)
2    DIRECT.ADDRESSING     = /IR3
3    AUTO.INDEXING         = (MA eq 000000001---)
4    NO.INDIRECT.OPERAND = (IR eq 10----------) + (IR eq 011---------)
5    SKIP                  = IR4 XOR (IR7*AC11 + IR6*(AC eq 0) + IR5*LINK)
6    AC=0                  = (AC eq 000000000000)
7    MB=0                  = (MB eq 000000000000)
8    MANSW = LDMA + LDMB + LDMEM + LDPC + LDIR + LDAC + DEP +EX + CONT + CLR
9-18 MANUAL SWITCH TESTS
```