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.

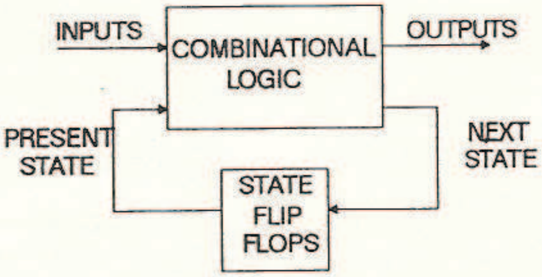

FIGURE 1 A CONVENTIONAL MODEL OF SEQUENTIAL SYSTEMS

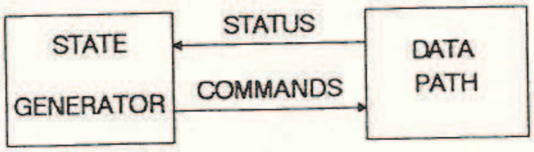

FIGURE 1 B FIRST DECOMPOSITION OF GENERAL DIGITAL SYSTEM

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.