Lecture #32 – April 28, 2004

The Bizarre Architecture of the PDP-8

In this lecture we will look at a computer architecture that was very popular in the late 1960s through the mid 1970s. In those days a university might have a single large computer system that serviced the entire campus, running administrative tasks during the working day and becoming free for students to use, experiment on, and hack after hours. This may be partly responsible for the tradition of programmers staying up all night working on their programs. In those days, for a research lab to have its own computer was a luxury. Any such machines as existed were relatively small minicomputers, usually much cheaper than the central mainframe and much less powerful.

It was into this market that Digital Equipment Corporation (DEC) introduced the PDP-8 in 1965, for the bargain basement price of around $30,000. By today's standards the PDP-8 is a joke, but it was used in many environments where a smallish machine could be programmed to perform a limited number of tasks in real time, such as process control, laboratory analysis, running experiments, annoying lab rats, etc.

PDP-8 Basics

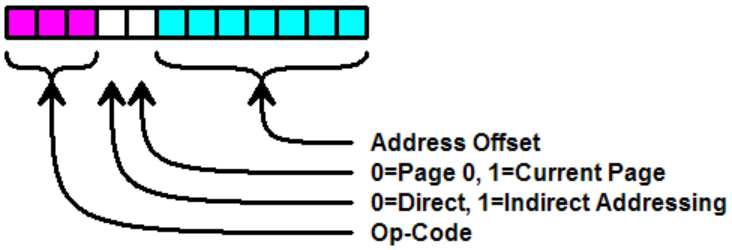
Several models were produced, each with slightly different capabilities. The basic model was a 12-bit machine, with 4096 12-bit words of ***magnetic core memory***, a 12-bit accumulator, and a 1-bit ***link register***. The link bit is *not* the same as the carry bit on modern systems; any carry out of an addition *complements* the link bit instead of simply setting it to 1. Programs and data were entered initially through a ***switch register*** on the front panel, and output was shown by the status lights showing the values of bits in the accumulator, link register, program counter, memory address register, etc. Advanced input-output could be accomplished through a 10 character-per-second Teletype™ machine with a paper tape reader/punch, and enhanced models of the PDP-8 might have one or two magnetic tape drives and a video screen.

Display systems in those days rarely had memory mapped video because of the high expense of memory. The act of plotting a random pixel on the screen caused the phosphor corresponding to that pixel to be energized *once*, and then over time the light output from the phosphor would decay to nothing. Programs would have to "revisit" each pixel repeatedly to keep them illuminated, and if the refresh rate was too low the image would flicker.

Memory on the PDP-8 is divided into 32 pages of 128 words per page. An instruction can address any word in the ***current page*** (where the instruction itself is stored) and any word in ***page 0***, but cannot reference an arbitrary word in another page directly. Instead, memory reference instructions allow for ***indirect addressing***, so that a referenced word may contain a 12-bit absolute address to an operand somewhere else in memory rather than the operand itself.

Memory Reference Instructions

All PDP-8 instructions contain a three-bit Op-Code. With just three bits for the Op-Code, there are only eight primary instructions. Six of those eight instructions are memory reference instructions, with one bit for direct versus indirect addressing, one bit for current page versus page 0, and the remaining seven bits for the address offset into the selected page. Those six instructions are formatted as shown in the following image,



Those six memory reference Op-Codes are as follows:

0: `AND` – Logical AND of operand with accumulator.
1: `TAD` – Two's Complement Addition of operand into accumulator.
2: `ISZ` – Increment operand and skip next instruction if result is zero.
3: `DCA` – Deposit (store into memory) and clear accumulator.
4: `JMS` – Jump to subroutine.
5: `JMP` – Unconditional jump to a memory address.

Several of these instructions have attributes worthy of discussion. Notice that while the `TAD` instruction does straight addition, there is no explicit load or store instruction. The `DCA` instruction *does* store the accumulator into memory, but it has the side effect of clearing the accumulator afterwards. If the accumulator is known to contain zero then a `TAD` instruction will perform the same function as would a load. Thus, to store the accumulator into memory and still keep the value in the accumulator a `DCA` instruction is always followed by a `TAD` of the same address. It is a bit clumsy, but with this technique two Op-Codes do the work of three.

There are no "traditional" conditional branches on the PDP-8 as there are on the 6502, 8088 or ARM processors. The `ISZ` instruction is used in that capacity. It increments (adds one to) the memory operand, then *skips the next instruction* if the result of the increment is zero. Skipping one instruction is very simple to implement in hardware. During the normal fetch-execute cycle an instruction is fetched and the program counter is incremented to the next word before the instruction is executed; if the `ISZ` detects that it needs to skip an instruction it need only increment the program counter once more.

In practice, If-Then-Else structures are implemented by following an `ISZ` instruction with a `JMP` (jump) instruction; the program flow either falls into the `JMP` and transfers control elsewhere or skips over the `JMP` and picks up with the following

instruction. If a memory variable must be incremented and you can guarantee that doing so will never result in zero (so the skip behavior is never triggered) then `ISZ` can be used as a simple memory increment instruction.

Subroutine calls on the PDP-8 use a mechanism that we have discussed before. Upon executing a `JMS` (jump to subroutine) instruction, the return address is stored into the first word of the subroutine and execution picks up with the second word. There is no explicit "return from subroutine" instruction. Instead, exiting a subroutine requires only that a `JMP` instruction jump *indirectly* through the first word of the subroutine. In essence, the `JMP` instruction doesn't know where it is going, but it knows who does.

```
SUB, 0          ← Word reserved for return address
     …          ← First executable instruction goes here
     …
     …
     JMP I SUB  ← Return from subroutine by jumping
                  indirectly through the first word.
```

This is a simple mechanism that is easy to implement in hardware, and any number of nested subroutine calls are allowed, but it is pretty obvious that recursion is not supported. Since memory is very tight on the PDP-8, some programmers were forced to use return address locations of subroutines as storage for temporary memory variables. This works only when the variable isn't needed at the same time the subroutine is called!

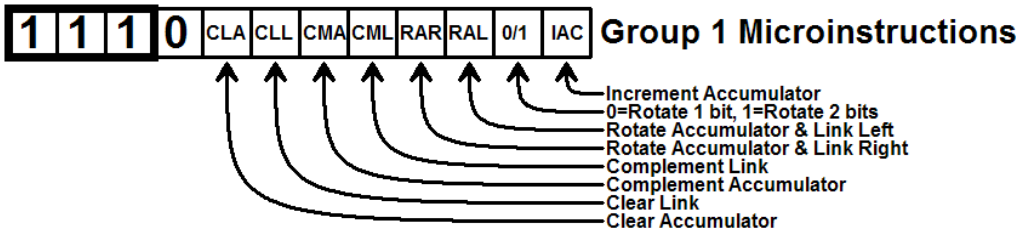
Microprogrammed Instructions

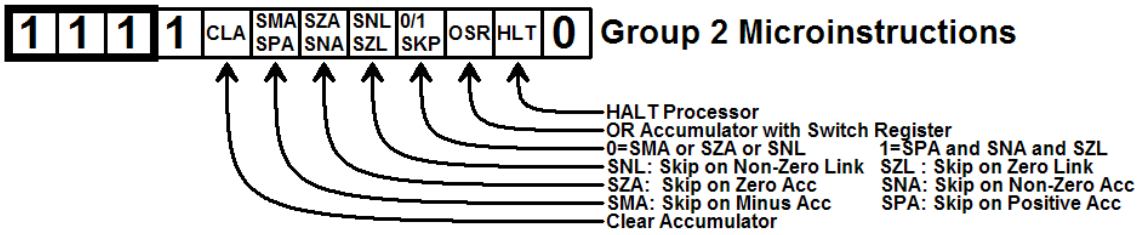
The remaining two Op-Codes correspond to what are called ***microprogrammed instructions***. These are instructions where each bit of the instruction (except for the Op-Code bits) has a function separate and unique from all the other bits. Composite instructions can be created by setting the bits for all desired functions. Upon execution, these instructions perform the functions in the order that their bits appear in the instruction word.

On the PDP-8, Op-Code 6 corresponds to input/output instructions. These are extensive, and differ depending on the model of PDP-8 being used and which peripheral devices (Teletypes, magnetic tape drives, modems, real-time clocks, etc.) are attached to it. We will not explore the details of Op-Code 6, but understanding how the rules work for Op-Code 7 applies here as well.

Op-Code 7 (what DEC called the ***operate microinstructions***) contains two groups of codes. A single bit distinguishes the group 1 microinstructions from the group 2 microinstructions, and the remaining eight bits of the word have (mostly) different functions depending upon which group is selected. The group 1 microinstructions are devoted to functions affecting the accumulator and link registers, while the group 2 microinstructions primarily implement a number of extra skip instructions roughly corresponding to what would be tests on status bits in other processors.

The layout of the two groups of microinstructions is shown below:





In the group 1 microinstructions you can see that there are simple ways of clearing the accumulator and link registers to zero, as well as ways of performing one's complements on each. Similarly, you can choose to rotate the accumulator and link register (as if they were a composite 13-bit register) either to the left or to the right, and by either one bit or by two bits. Finally, you can increment the current value in the accumulator. In the assembler, you create a composite microinstruction by specifying all the desired micro-Op-Codes on the same line. For example, to clear the accumulator and link registers and increment the accumulator (effectively moving the constant 1 into the accumulator), you would type `CLA CLL IAC` on the same line, and the assembler will build the appropriate instruction. Some microinstructions must be performed before others; it doesn't make sense to clear the accumulator after you increment its value, but clearing before incrementing *does* make sense. Illegal or meaningless combinations are often possible in microinstructions; it isn't clear what happens if an instruction is executed that corresponds to `RAR RAL` (rotate right and left simultaneously).

The group 2 microinstructions mostly deal with new types of skip instructions, but notice that there is a `CLA` function here as in group 1. Combined with the `OSR` instruction that performs a logical OR between the accumulator and the 12 input switches on the front console, the combination `CLA OSR` reads the switches (which were often used as game buttons).

The PDP-8 has no status register as we are familiar with the concept. On modern processors you perform an operation such as add or subtract and then sometime later check the status bits to determine if the result was zero or negative. If intermediate instructions don't affect the status bits then the status values will propagate forward in time to the point where they are checked. On the PDP-8 you check what the accumulator and link registers contain *right now* and skip the next instruction appropriately; is the accumulator's <u>current</u> value zero or negative? In some sense the PDP-8 asks what is happening now, and modern processors ask about what happened in the past.

The fourth bit from the right controls whether the meaning of three other bits is `SMA/SZA/SNL` (skip on minus accumulator, zero accumulator, or non-zero link) or `SPA/SNA/SZL` (skip on positive accumulator, non-zero accumulator, or zero link). These roughly correspond to the N, Z, and C status bits on modern processors.

The skip bits can be combined to create composite skips, but in peculiar ways. If the `SMA/SZA/SNL` group is chosen, then the selected functions are OR-ed together. The instruction combination `SMA SZA` skips the next instruction if either the accumulator is negative or the accumulator is zero (a signed integer less than or equal to zero). If the `SPA/SNA/SZL` group is chosen the functions are AND-ed together. The instruction combination `SPA SNA` skips the next instruction if the accumulator is positive and the accumulator is non-zero (a signed integer greater than zero). A special case where the control bit equals 1 but no skip bits are selected is treated as an *unconditional* skip.

Conclusions

Programming the PDP-8 was not especially easy. In a time when hardware was *very* expensive, the engineering decisions made in its design were strongly geared towards minimizing overall costs. This meant taking a considerable number of shortcuts in the instruction set and thereby increasing the workload on the part of the programmers.

As an example of the cost-benefit analysis done at the time, consider the memory systems for a minute. It seems odd to us now that memory words on the PDP-8 are so narrow (12 bits) when today we are used to instructions and memory systems which are 32 bits wide, 64 bits wide, or even wider. At the time of its design, the most expensive components were the driver circuits for the magnetic core memory, and a driver circuit for an address line cost about the same as a driver circuit for a bit line. Thus, for a memory system containing 2^N words of N bits per word (N=12 on the PDP-8) it was roughly the same cost to *double* the number of narrow memory words as it was to increase the width of memory by *one bit*! So, if you had a memory system containing 4K 12-bit words, adding one driver circuit gives you the choice between 4K 13-bit words or 8K 12-bit words. Most people would decide on doubling the overall number of narrow words so they could pack that many more instructions into their programs.

All processor instruction sets are compromises between overall functionality and instruction complexity (or cost). When you are forced to have a narrow memory word, the compromises force more extreme engineering decisions. A prime example of this on the PDP-8 is the selection of the `TAD/DCA` combination instead of explicit load, store, and addition instructions. Another is the use of skip instructions instead of conditional branches; they are easier to implement in hardware and do not require an address or offset as part of the instruction (although I always thought a "skip two instructions" mode would have made more sense). It is also instructive to examine a processor model that does not use explicit status flags. Finally, the use of microprogramming allows a programmer to pack a lot of different but often related commands into one instruction.

While the PDP-8 is obsolete today, and exists mostly in software emulations, it still offers insights into hardware and instruction set design worthy of contemplation.