Plan your next getaway
Family retreats in the great outdoors
Vrbo
OPEN

A brief tour of the PDP-11, the most influential minicomputer of all time

It helped popularize the interactive computing paradigm we take for granted today

Benjamin Cassidy - Senior Writer

Vintage

The history of computing could arguably be divided into three eras: that of mainframes, minicomputers, and microcomputers. Mainframes were the first computers to be commercially produced, and they were followed by minicomputers, which were followed by microcomputers. The last two influential eras are discussed in this article.

In their moment, minicomputers were used in a variety of applications. They served as communications controllers, industrial control systems, and laboratory data processors. Many of these applications, in turn, made it possible for them to pioneer the interactive computing paradigm we take for granted today. Some of the most influential minicomputers were the PDP-11 series, which was more or less the pinnacle of the minicomputer era.

In order to understand why the PDP-11 was influential, it will be grounded for an instance of minicomputers that would make a new market for themselves by being readily available, expanding and adapting to meet customer needs. In contrasting the PDP-11 to Digital's both the DEC-operating system and the Programming language, it would also greatly influence the next generation of computers. Finally, we will take a look at the 21st Reason of the PDP-11 in a secure universe of the computer's remarkable, creative, and influential legacy.

Early PDP-11 models were not overly impressive. The first PDP-11 11/20 was a 16-bit machine that could only address 8KB of RAM, had no mass storage and had an KHz-20 bespoke printer connection that proved unreliable. Nevertheless, the 11/20 was a hit with customers, who appreciated its affordable price, its compact size, and its 1 MHz clock speed. A 21 MHz clock speed, and a Thomas UNIBUS hardware bus that would support future hardware peripherals. This was a winning combination for its creator, Digital Equipment Corporation.

The initial application for the PDP-11 included real-time hardware control, factory automation, and data processing. As time went on, the PDP-11 was used in a variety of more specialized applications, such as in medical imaging, laboratory data processing, air traffic control, nuclear power plant control, and telecommunications. It also powered the word processing and data processing that we now take for granted.

Assembler programming basics

Before high-level languages such as Python, Java, and Fortran were invented, programming was done in assembly language. Assembly language is a low-level programming language that is closely tied to machine language. Instructions for the processor are written in assembly language.

Assembly language is a low-level programming format that is connected to machine language. Instructions for the processor are written in assembly language, and the hardware uses a set of instructions corresponding to each of the processor's architecture. Simply put, assembly programming means your data type by hand, and it means that you must be aware of the processor's architecture. Assembly language is also known as symbolic machine language.

Keywords	Translation
ADD, SUB, MUL, DIV	Add, subtract, multiply, and divide are employed for the operating system's use
AND	Logical conjunction operation (AND) is performed
JMP, INPUT, RET, OUT, CMP, JZ	Flags the overall status of memory locations RET, OUT, INPUT, JZ, and CMP, respectively
MOV, MVI, LDA, STA, LXI, PUSH, POP	Data is moved from one place to another
DAA, RLC, RRC	Sort of data segment arrives: hexadecimal, binary data
HLT, nop, ei, di	Sort of data segment arrives: hexadecimal, binary data

Although numeric values can be used for memory addresses, the use of labels instead of hard-coded addresses makes it easier to program and makes the code relocatable. Comments can be added to assembly language programs to make coding easier, and comments begin with a semicolon.

The data assembler directive puts the data in a memory segment that's both readable and writable. The memory reserved for code is read-only to prevent programming errors from corrupting the program and causing crashes. This segment is also known as the .text segment. The .bss segment is used for data that's not initialized. The .bss segment is the feature also doubles the address space by enabling both for code and RAM for data—this was considered quite an innovation at the time. Accordingly, Intel's 8086 microprocessor later made extensive use of segments.

BOOK NOW
Hotels.com

An original front panel of a PDP-11/05.

PDP-11 architecture

What about the PDP-11 made it better to program with its simple but powerful architecture.

Eight 16-bit registers were the heart of the instruction set architecture, or ISA. Six registers were general-purpose, one was the current, and one was the program counter. Registers could be used in many ways, including as stack pointers as well as for temporary storage, counting, and accumulating. The PDP-11's ISA was also orthogonal, meaning that instructions could be used with any register. Even a number of unconditional jumps and branches.

The PDP-11 ISA was also orthogonal in that it was consistent in the ISA's instruction set. Instructions could be used with any register. This is also called "a free jump" because the effective address of "STARTT" uses all 16 bits. A long jump is also an example of a direct address.

BR CALC ; Branch to the offset address of CALC, forward up to 127 words or backward up to 128 words

JMP CALC ; Jump to the address CALC, 32 words forward or backward up to 128 words

The branch instruction is also called a "short jump" because you can't jump as far. In this case, CALC is an indirect address, and the jump can go anywhere in the address space. The address of CALC is located in memory, and the memory, the equivalent instruction would be to add 32 words to the register PC.

Why the need for a long and a short jump? One word: memory. The 16-bit address in the long jump uses an extra byte. In a program or operating system with a lot of branching, this can add up to a sizable fraction of your 8K or 32kwn RAM. This is another example of clever PDP-11 technique to deal with memory.

Only whole vacation homes. All in one place.
Villa Tacky are always posh. So you only stay in the best of your people.
Vrbo

Testing and branching

Testing and branching instructions are fundamental to conditional and looping

Comparison, math, and logic instructions perform operations on data that result in an update to the condition code, or CC. The CC is a special-purpose register that stores the ISA's current status. Its components are CC contains the Carry, Overflow, Zero (or negative) bit, the 1 or Overflow bit, and the C or carry bit. Branches and jumps can be conditional or unconditional. The CC is after an operation that sets the CC, such as CMP.

Here's an example:

CMP A,B ;Compare A and B by subtracting B from A, and setting C, N, V, Z bits accordingly

BEQ CALF ;If they are equal, short branch to location CALF

And here is the assembly language equivalent of the following loop:

for (x=0; x < 100; x++)

{ y=y+5;

CLR R0 ;Clear, or initialize register 0 to zero

CLR R1 ;Initialize register 1 to zero

ADD#5 ; ADDR is a memory label to the memory address

CMP R0, R100 ;Compare register 0 to the value 100

BGT QUIT ;If register 0 is greater than 100, branch to QUIT

ADD R0,R1 ;add register 0 to register 1

INC R0 ;add 1 to register 0

BR ADDR1 ;Branch to ADDR

QUIT : QUIT is a memory label in this memory address

TST can be used to check a single value:

TST A ;Test the value of A to see what it is 0 or N, depending on the data

And the PDP-11 has a full complement of conditional branch instructions, including but not limited to:

BEQ ;Branch if the previous result is equal to zero, that is, if Z bit is set to 1

BNE ;Branch if the previous result is not equal to zero, that is, if Z bit is zero

BGT ;Branch if the first operand is greater than the second

BGE ;Branch if the first operand is greater than or equal to the second

BLT ;Branch if the first operand is less than the second

BLE ;Branch if the first operand is less than or equal to the second

Data operations

Let's take a look at instructions for data operations:

MOV X, R0 ;Move the value 1 into register 0

MOV R0 ,R1 ;Move the value 1 into the memory location R1

SXT, SWAB, ASR, ASL, ROR, ROL, SBC, ADC, SUB, CMP, BIT, BIC, BIS, ADD, MOV

SXT: Sets X, 0 if there has to calculate and initialize, 1 if pass, and 1 if pass, with the address of R1. The assembler replaces the symbol R1 with its address. It calculates the address of X relative to R1. The symbolic address can be relocated in memory by the linker when the program is linked into an executable.

ADD X, R1 ;Add the value 1 to the contents of register R1

ADD X, R1 ;1 add 1 to the value in memory location R1

SWAB: Swaps the high and low bytes of a 16-bit word

In almost all computer instructions, registers and memory location can be used interchangeably. This gives the assembly programmer more freedom, and it's one key thing to keep track of manually. It is helpful to keep data in registers whenever possible, as every register instruction that replaces a memory address reduces the memory footprint of the program. Keep in mind that the overhead of assembling programming even high-level languages—assembler code almost always runs faster and uses less memory.

BOOK NOW
Hotels.com

Two's complement notation and negative numbers

The PDP-11 natively supported positive and negative numbers through 16-bit and 8-bit integers. Later models added floating-point numbers.

In this way, the arithmetic logic unit (ALU) can add, subtract, multiply, and divide positive and negative integer numbers. Two's complement is a clever way to represent negative numbers. Two's complement makes the ALU's job easier, and it's also fast.

Traditionally, the PDP-11 used an Intel 8008 microprocessor, which was 8-bits, 0/255 represents zero and in the same all 8 bits, or 11111111 in binary. Since we are programming under 16bit with the PDP-11, we can have 16.

Using subroutines

Let's go deeper by working through some situations on a PDP-11. Say you're writing an assembler program and you have a piece of code you will use repeatedly from different parts of your program. Why not write a subroutine and call it from your program? Subroutines are a key part of programming in the early days of computing, and they are still important today.

The trickiest part of getting back to where you left off in the subroutine. The architecture's stack can be used to manually save your address, or the PDP-11 can do it automatically.

MOV PC, R07 ;ADDR: Save the current address in register 7 to the memory location RET_ADDR

; and then copy back at the end of your subroutine MOV R07, ;RET_ADDR_PC

JSR PC, ;SUBROUTINE ;Jump to subroutine, which is a new subroutine, calling another subroutine! The bookkeeping and complexity add up quickly, which could lead to unreliable and buggy code. The PDP-11 had a fix for this issue that accounted for some of the time and memory: the JSR and RTS pair of instructions.

JSR PC,SYMBOL ;Jump to subroutine SYMBOL ;save the program counter value onto the stack

The stack is a special-purpose register. Only 16 bits. It pushes the value of the program counter onto the stack memory location of the stack pointer. The stack pointer is also a special-purpose register. It's called the stack because it's like a stack of CD's. The beginning of your subroutine. At the end of your subroutine, you execute the statement:

RTS PC, ;Return from subroutine, using PC

This pops the return memory address off the stack and copies the address into the program counter, thus jumping to the return address. There's lots of stack operations involved and for nested subroutines. You do have to coordinate your stack pop operations with the stack push operations, though, because one mix and you'll have a serious programming error.

Consider this example of a subroutine. Let's say we want to convert from degrees Fahrenheit to degrees Celsius for 100 values. Here's the code:

```
; The code to C F
;
Start:
Set MAX(R0);
For (i=R0; i< 100; i++)
{
MOV(i) - FTMP(i);
;
Call FDEGCELSIUS
;
Mov FTMP(i),(i)
Return (i)}
;
; Convert F to C, JSR from Fahrenheit to Celsius
;
CLR F ;Initialize the degrees counter
;
ADAPH:
CMP R100,R0
;
BGTE QUIT ;If register 0 greater than or equal to 100, call loop
;
PUSH R1 ;Push degrees F in value onto stack, as a parameter for FTOC subroutine
;
JMP PC, FTOC ;Call FTOC subroutine
;
POP R1 ;Retrieve Centigrade temperature
;
MOV R0, MEL(R1) ;Copy Centigrade value to temp in memory
;
INC R1 ;Increment R1
;
BR ADAPH ;Loop again
;
QUIT:
; ;The rest of your program could go here
;
HALT
;
FTOC: ;Subroutine to convert degrees Fahrenheit to Celsius
;
PSP R0
;
SUB R0,R0
;
MUL R0
;
DIV R0
;
Push R0
;
RTS PC,R5
;
;MUL
;
MUL: MOVB(R0)-(SP) ;Clear array of values
```

And here's another code example that shows the power of the auto-increment deferred addressing mode. In this example, we're trying to implement the operation of a particular algorithm to efficiently move characters from one location in memory to another. where they can be printed.

```
;
;
;
```

MOV $10, R2

MUL R2,R0

System instructions and breakpoint traps

The PDP-11 has assorted other system instructions, including:

HALT / Halt the CPU

WAIT / Wait for an interrupt

RESET / Reset the external bus, including hardware device control and status registers

NOP / Do nothing ("No Operation")

Such system instructions are used by debuggers:

TRAP / User invoked interrupt

BPT / Breakpoint trap

IOT / Input/output trap

RTI / Return from interrupt

Trap instructions allow a computer to park whatever it is doing in a safe place, process something else, and then resume the original processing where it left off. This is useful for debugging or processing IO, priority system calls, or user-defined subroutines. The BPT and RTI instructions support the basics needed for building a debugger. ADB was the first popular UNIX debugger and is the direct predecessor of GDB, the powerful GNU debugger. ADB can set breakpoints, display values in registers and memory, set values, and single-step through code. Although sparse compared to GDB, it was a programmer's best friend in the '70s and '80s.

How does debugging work at the machine code level? When the BPT instruction is run, the trap bit in the processor status register will direct the processor to a certain program address after executing each instruction. This allows the debugger to single-step through program memory one instruction at a time. It also allows you to set breakpoints in your code so you can stop to a label in debugging and then step to examine the values.

To try this on any UNIX or Linux program with ADB or GDB, start a program in the debugger: `adb a.out`. And then single-step by typing `:s`. The `:P` is using breakpoint instructions.

Hello, World

Here is a program that interacts with the operating system and uses system calls to write output:

```
/ Hello, World in PDP-11 Assembler

.text

        sys write; msg; 15 / call UNIX system call, write to print msg text

        sys exit

.data

msg:
```

We call the system write routine with the address of the "Hello, World" string, denoted as "msg." We use the `.data`assembler directive to tell the assembler that the address of msg is in the data segment.

The "Hello, World" text is defined as a sequence of bytes followed by a null character. Its memory location is defined by the label "msg." So we print the text by pushing the value of the program counter register onto the stack, pushing the address of the text onto the stack, and jumping to the address of the write routine. The write routine pops the address of the text off the stack and prints the text. The write routine returns to the caller's address by executing the RTS instruction which copies the return address of the stack into the program counter.

This is how subroutines are called and how many subroutines can be nested. It's also the underlying mechanism for how high-level languages nest subroutines.

Booting a PDP-11 simulator

To really understand what the PDP-11 was capable of, let's run some of this code on a simulated PDP-11. SimH, an amazing computer simulator application, is a veritable time machine into computing's past. It allows you to run in simulation the code of dozens of classic computers, including several types of PDP-11s. We will use SimH, scripts, and boot data to configure a PDP-11 running an early version of BSD UNIX. On this, we can compile and run some assembly program examples. This version of UNIX was popular with universities and was used to teach classes, such as PDP-11 Assembly Language. These instructions assume installation on Linux but can be adapted for other platforms. In fact, I'm running 2.11 BSD UNIX in SimH in Ubuntu Linux running in Windows. I'm using Oracle's VM Virtual Box to run Ubuntu Linux under Windows and running SimH under Ubuntu.

We will be using Warner Losh's SimH configuration for a PDP-11/93 system with 4MB of RAM, two RD54 disk drives, one TS11 tape drive, and a generic DZ11 serial port. Warner pathologically recreated the boot media for the last and best UNIX on the last and best PDP-11 he provides a detailed description of the UNIX boot process, too). This version of UNIX actually supports Ethernet, so an ambitious person could even put a simulated PDP-11 on the web.

Let's start our configuration. First, boot into Ubuntu. Then, bring up the Terminal application to access the command line.

Build the latest version of SimH for PDP-11:

```
$ mkdir sim
$ cd sim
$ sudo apt install git
$ sudo apt install gcc
$ wget http://sim.trailing-edge.com/sources/simh311-B.zip
$ unzip simh311-B.zip
$ make pdp11
$ sudo cp BIN/pdp11 /usr/local/bin

Now retrieve and assemble the boot image for 2.11 BSD UNIX patch 195:
$ sudo apt install git
$ sudo apt install expect

$ mkdir mk211bsd
$ git clone https://github.com/bsdimp/mk211bsd
$ cd mk211bsd/195
$ wget https://www.tuhs.org/Archive/Distributions/UCB/2.11BSD-pl195.tar
$ tar xvf 2.11BSD-pl195.tar
$ gunzip *.gz
$ perl mk211bsd195.pm pl
$ expect 211bsd-195.expect  #

Edit the last two lines of 211bsd-195.ini to remove the #

Start the simulator:

$ pdp11  211bsd-195.ini

: roCR,RQunix
: ^D

Login as root with no password:
login: root
[1] root+> df
```

If you made it this far, congratulations—you have assembled and booted into the last, best version of UNIX for the PDP-11, released in 1992.

Enlarge

Now that we have configured and booted into an emulated PDP-11 running classical UNIX, let's write and run a basic assembly program.

```
First, get your backspace key to work by typing:
stty erase

Type the backspace key where it says . Then

mkdir coding

cd coding
```

Next, boost your tech street cred by running your antique version of the vi editor and create a makefile that compiles and links the program into an executable:

```
vi makefile

A%= /bin/as

SEPFLAG= -i -A

All: hw.os

hw: hw.o

    ld $(SEPFLAG) -o $@ /lib/crt0.o hw.o -lc

 key to end insert mode:

 :

 :
```

Next, type in your assembler program:

```
vi hw.os

: to insert text:

[assembly program goes here]

 :
```

Type "make" to compile and link your program. Now run your program by typing `./hw`.

By writing and running code on a simulated PDP-11, you can relive the golden age of the minicomputer. Fifty years ago—before cell phones, before personal computers, before bitmapped screens and mice—this is how you wrote code.

Page: 1 2 3 4 5 6 7 Next →

READER COMMENTS 419 SHARE THIS STORY

Advertisement

CHANNEL ars

SITREP: F-16 replacement search a signal of F-35 fail?
Footage courtesy of Dvids, Boeing, and The United States Navy.

SITREP: F-16 replacement search a signal of F-35 fail?
Sitrep: Boeing 707
Steve Burke of GamersNexus Reacts To Their Top 1000 Comments On YouTube
More videos

← PREVIOUS STORY NEXT STORY →

Related Stories

High performance computing on gamer PCs, Part 2: the software choices
Bastion's demise approaches Intel to stop shipments in mid-2021
How has an increase in system complexity affected new programmers?
Are Why are some programming languages faster than others?

Sponsored Links by Taboola

Bone On Bone? These "Bionic" Knees Have Will Transform Your... ★★★★★
1 Simple Trick To Cut Your Electric Bill by 10% (Try... ★★★★★
NFL Star Rob Gronkowski's Favorite Shoes Well & Good/Sponsored
Top 5 Solar Providers in... Solar Energy
3 Toxic Foods for Dogs: the One Meat You Should Never Feed You... ★★★★★
If You Have US Dollars In Your Bank Account ... Goldmoney.com

CCPA Notice

Today on Ars

Hackers exploit WordPress plugin flaw that gives full control of millions of sites
Twitter posts the code it claims determines which tweets people see, and why
GM kills more than CarPlay support, it kills choice
Google Bard gets better at homework with improved math and logic capabilities
Deadly Fungal outbreak in Wisconsin linked to neighborhood construction
Review: DEO: Honor Among Thieves is a worthy homage to the classic RPG
A remotely operated lab is taking shape 2.5 km under the sea
Right to repair, universal charging port mandates enjoy a rare Canadian money

STORE
SUBSCRIBE
ABOUT US
RSS FEEDS
VIEW MOBILE SITE

CONTACT US
STAFF
ADVERTISE WITH US
REPRINTS

NEWSLETTER SIGNUP

Join the Ars Orbital Transmission mailing list to get weekly updates delivered to your inbox.

SIGN ME UP! →

CONDÉ NAST

CNMN Collection
WIRED Media Group
© 2023 Condé Nast. All rights reserved. Use of and/or registration on any portion of this site constitutes acceptance of our User Agreement (effective 1/2/2020) and Privacy Policy and Cookie Statement (effective 1/2/2020). Ars Technica Advertiser Choice is a trademark of Condé Nast. Ars Technica may receive compensation for some links to products and services on this website. Read our affiliate link policy.
Your California Privacy Rights | Your Privacy Choices
No part of this site may be reproduced, distributed, transmitted, cached or otherwise used, except with the prior written permission of Condé Nast.
Ad Choices