CS220 Computer Organization

Spring 2012

Machine Problem 3, Assembly Language Option

Due February 24, 2012

Assembly Language Option

If you are fairly comfortable with PDP-8 assembly language, do this option. If you are still struggling with programming in assembly language, or understanding the operation of the PDP-8, do the Java option.

Objectives

The objective of MP3 is to give you first-hand experience with computer arithmetic at the machine level. Although some details of PDP-8 arithmetic are obsolete, most of the principles involved apply to almost any machine. For example, modern computers no longer use specialized AC and MQ registers for integer multiply and divide. General purpose registers are used instead, and those registers are typically 32 bits wide, not 12. The nature of the registers involved, however, is a fairly minor point. Modern integer multiply, divide, and shift instructions are similar to the PDP-8's Extended Arithmetic Element (EAE) instructions in almost all ways that are important.

To keep the arithmetic work somewhat interesting, and also to learn some useful methods, MP3 will calculate *e*, the base of the natural logarithms, to extremely high precision (129 digits, or more if you want), much higher than one could do using double-precision floating point operations. The entire program to calculate *e* in assembly language on the PDP-8 is too complex to do in a one-week assignment, so we will focus on one particular subroutine, the one that does arbitrary precision integer multiply.

How to Calculate *e* to as Many Digits as You Want

The following method is not the most sophisticated, but it's probably the simplest. Here is the standard series expansion for *e*:

$$e = \sum_{i=0}^{\infty} \frac{1}{i!} = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \cdots$$

Considering just the first n+1 terms, multiply top and bottom of each ratio in the series by n!:

$$e \approx \frac{1 + n + n(n-1) + n(n-1)(n-2) + \cdots + n!}{n!} \approx \frac{a}{b}$$

Now we have *e* as the ratio of two integers, *a*/*b*. Very large integers, mind you, but large integers are easy to handle and are subject to no round-off error—the numerator and denominator are exact to the last bit. All one needs to calculate them is arbitrary precision add and multiply.

Having an approximation for *e* as the ratio of two large integers *a*/*b*, the next step is to convert *a*/*b* to a decimal fraction so it can be printed. If you just do an integer divide of *a* by *b*, you'll get "2", since e ≈ 2.7 and integer divide truncates (discards the fraction). That's one digit. If you do the integer divide (10*a*)/*b*, you'll get 27, two digits. You see where this is going. In general, to get *k* digits:

$$\frac{10^{k-1}a}{b}$$

So all we need do is multiply *a* by say 10^{128}, then divide by *b*, and the result will be a binary integer that represents the first 129 digits of the decimal fraction corresponding to our ratio. If we then print that integer, and if we've used a sufficient number of terms in the series (i.e. *n* is sufficiently large), the printout will be the first 129 digits of *e*. Note that 10^{128} is 28 orders of magnitude larger than a google, 10^{100}. Ever think you'd need such a large number?

So in addition to arbitrary precision integer add and multiply, we also need divide, and we need a method for printing an arbitrary precision integer in decimal. Converting binary to decimal also relies on integer divide, specifically dividing by 10 repeatedly, where the remainders of those divisions are the digits we want (but in reverse order). In practice I use a slightly more sophisticated method for converting to decimal, dividing repeatedly by 1000 to get triplets of digits, because dividing arbitrary precision numbers by 1000 is just as fast as by 10, and 1000 is the largest power of 10 that fits in 12 bits.

How many bits do we need for our integers, and how big should *n* be? $\log_2(10^{128})$ is about 425.2, so that's about how many bits we need. 36 12-bit words is 432 bits, so 36 words should do it. 85! is about 2.8×10^{128}, slightly above the precision we need, so $n = 85$ should do it. Once we've got correct arbitrary precision add, multiply, and divide, we can make the integers as big as we want.

Reference Documents

The reference documents are the same as for MP2, but I've indicated different pages in some cases.

- *Small Computer Handbook* (1970): Pages 58 – 63 on the EAE instructions.
- *OS/8 Handbook* (1974) Chapter 3: Pages 3-5, starting at "Character Set", to 3-30, ending at "Conditional Assembly Pseudo-Operators", but skipping the sections "Symbol Table", "Internal Symbol Representation for PAL8", "Autoindexing", "Extended Memory", and "End-Of-File". This is DEC's documentation of the PAL8 assembler.
- *PDP-8 Sim*: This explains operation of the simulator. **Pay particular attention to "Hardware Breakpoint"**.

- *PAL8S*: This explains how to use the PAL8 assembler that comes with the PDP-8/I simulator, and the differences between PAL8 as implemented and DEC's original.

Source Code

MP3 includes these PAL8 source files:

`e`	Main program for calculating and printing *e*. Defines the number of bits to use and *n*. Calculates and prints the ratio by calling subroutines in other files.
`bigutil`	Utility subroutines, including argument fetching, a bignum stack, and routines for copying and initializing bignums.
`bigadd`	Subroutines for adding, negating, and subtracting bignums.
`bigmulxx`	Multiply subroutine. This is the one on which you will work. Details in the next section.
`bigdiv`	Divide subroutine. Very simple and slow, one bit at a time. Have a look and see if you understand what it's doing.
`bigprint`	Print bignum in decimal, and print ratio of two bignums as a decimal fraction.
`tty`	Teletype I/O utilities that I wrote for my version of MP2.

These files are a complete, functioning program that calculates *e* to 129 digits. They should work in any order, but I've tested them only in the order given above.

Procedure

If you have installed the simulator on your own computer, get the latest version as instructed in the *PDP-8/I Simulator* section on Blackboard. There are some enhancements and bug fixes that you might need. If you are using lab (224) computers, I will install the latest version by 3:30 on Friday, Feb 17. If you do Help→About you'll see the version number, which should be 3.2.

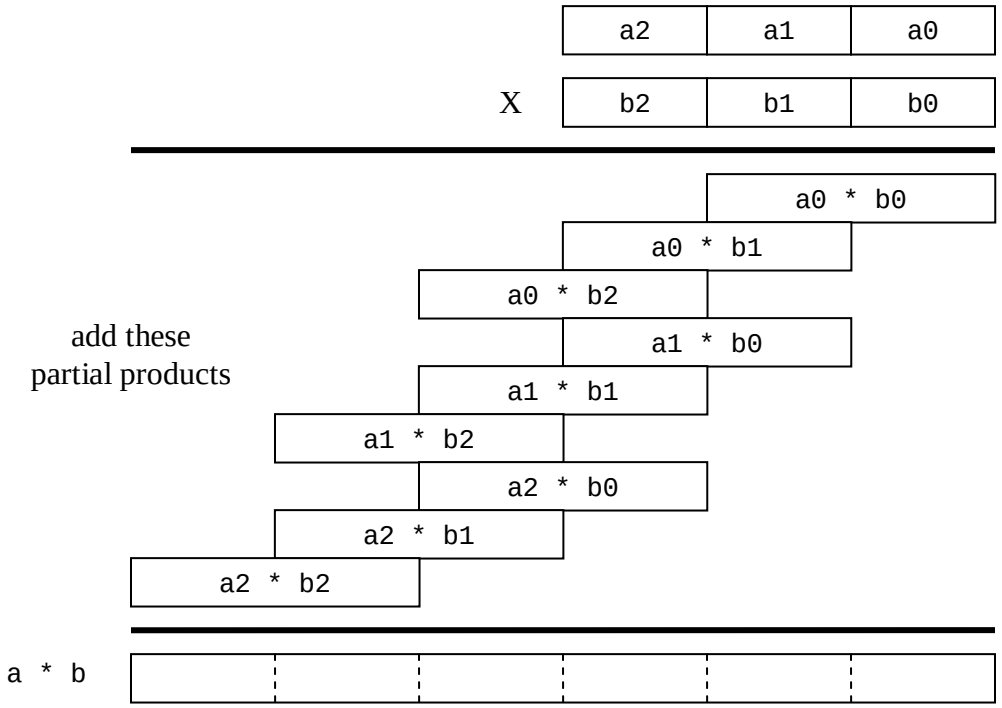
Load and run the PAL8 source files to confirm that you have a properly working program.

Look at `bigmulxx.pal8`. You'll notice that the `bigmul` subroutine provides a basic framework for multiplying by calling utility routines for fetching arguments, allocating a place to put the product, and returning the product to the caller. You'll also notice that all of the actual multiplication is done by the subroutine `multiply`, which is included in octal machine language, not assembly language. I did this so you have a working version of the program, but you can't easily see how the code works. Your goal is to replace my machine language version with your own assembly language version.

Start by creating a copy of `bigmulxx.pal8`, calling the copy `bigmul.pal8`. You can then work on the copy without losing the original.

The first thing to note is that bignum multiply, like the EAE MUY instruction, takes two single-precision operands and produces a double-precision result. For MUY, single precision means one 12-bit word and double precision means two 12-bit words (modern integer multiply instructions work the same way, typically on 32-bit words). For bignum, single precision means `bigWords` consecutive 12-bit words in memory (`12*bigWords` bits), where `bigWords` is a symbol defined in `e.pal8`. Double precision means `2*bigWords` words, or `24*bigWords` bits. Multiply cannot overflow because there are enough bits in the product for any multiplication. Note that there are both single- and double-precision bignum add subroutines in `bigadd.pal8`.

Bignum multiplication is done by using the MUY instruction to multiply all pairs of 12-bit words from the source operands and adding the 24-bit partial products to a double-precision bignum at the appropriate word offset. For example, if `bigWords = 3`:



The key to adding the partial products is getting the carry propagation right.

You can use any of the other subroutines in the source files, but you don't have to. For computing *e* all you need is unsigned (natural binary) multiplication, but my version does a signed multiply using a congruence formula modulo $2^{12*bigWords}$.

You have probably realized by now that you might be able to convert my octal machine language version back to assembly language to see how it works. Converting machine language to assembly is called *disassembly*, and studying object code to learn how it works is an example of *reverse engineering*. I have no objection to you trying to reverse engineer my multiply routine, but keep in mind that any disassembly you produce will have no comments and no symbolic names for variables, so it's not that easy to understand. You are probably better off spending your time figuring out how to multiply.

Deliverables (Blackboard)

- Your PAL8 `bigmul` source file.

If you want to send me more than one file, put them in a ZIP folder. Do not submit multiple files except in a ZIP folder. Please don't use RAR, it's too hard for me to read. If you want to add separate comments to me, put them in a document in a ZIP folder along with your code. The filename of the one file that you submit, either PAL8 or ZIP, should start with your last name. If you follow these rules, I'll be in a better mood when I grade your work.