Four Architectures – One Application

Brian J. Shelburne
Wittenberg University
Dept of Mathematics and Computer Science
Springfield Ohio 45501
937-327-7862

bshelburne@wittenberg.edu

ABSTRACT

Like many computer organization courses Wittenberg's Comp 255: Principles of Computer Organization has an assembler language component. Assembler provides a language to describe architectural features and assembler programming assignments allow students to work directly with those features. Unfortunately, no single computer architecture embodies all the possible architectural features one might wish to present. The solution I've adopted is to cover multiple architectures *and their assemblers* which allow a wider range of features to be covered. To compensate for the added burden of learning multiple assemblers, programming assignments are based on a common application. This allows the student to concentrate on the particulars of the assembler and its underlying architecture. The common application is carefully designed to span a range of standard programming approaches or techniques. Programming one application in four assemblers highlights the similarities and difference of the four underlying architectures.

Categories and Subject Descriptors

C.1.[Processor Architectures]; D.3.2 [Programming Languages] Macro and Assembly Languages

General Terms

Algorithms, Design, Languages

Keywords

Computer Organization, Assembler, PDP-8, RISC Architectures, Stack Architectures, Java Byte code, Intel 80x86, Simulator

1. INTRODUCTION

Wittenberg's COMP 255: Principles of Computer Organization course is a sophomore level course usually taken after the student has completed an introductory programming course. Textbooks for the course have been the "standards": Tanenbaum's *Structured Computer Organization* [6], Stalling's *Computer Organization & Architecture* [5] and currently Murdocca & Heurling's *Computer Architecture and Organization* [2]. The teaching of assembler has always been integral to such courses as witnessed by the fact that a number of textbooks on computer organization come bundled with simulators that allow students to create, edit, and run assembler language programs for the architectures discussed in the text [2], [3], [7]. The pedagogical use of these simulators is obvious: algorithms can be implemented using the assembler of the architectures and students can write programs on what they learn [8] [9] [10].

Unfortunately no single computer architecture can embody all possible architectural features: register vs. stack architectures, RISC vs. CISC-like features, addressing modes, and instruction mixes etc. The options are to either carefully choose/design an architecture that covers the most important architectural features or to study multiple computer architectures/assemblers I have chosen the latter course of action.

The decision to cover multiple architectures was one that evolved over time. What originally was a course in VAX assembler ([1]) was augmented by the introduction of PDP-8 assembler. This was done because I had built a simple PDP-8 simulator program [4] and the simpler PDP-8 architecture provided a gentler introduction to assembler programming. VAX assembler gave way to Intel 80x86assembler. Textbooks used for the course introduced other architectures and simulators. Tanenbaum's *Structured Computer Organization 4th Ed.* [6] introduced a micro-programmable simulator which could execute a subset of Java Byte code. Murdocca and Heuring's textbook [2] introduced an architecture/simulator called the ARC (short for A Risc Computer) which was based on SPARC architecture.

By this time four very different computer architectures were incorporated into the course and to varying degrees studied, compared and contrasted. Software in the form of simulators or assemblers allowed students to write and execute assembler language programs for each of the four architectures. By the end of the course students had written assembler programs for the PDP-8, the RISC-like ARC architecture, a stack architecture based on Java Byte Code, and the Intel 80x86.

Having four architectures to work with permitted flexibility in generating interesting and contrasting approaches when examining different architectural issues. To compensate for the additional time required to cover the details of four different assemblers, a common programming application was the basis for many labs and assignments. The application had to be *interesting* in that it required non-trivial use of programming techniques involving control structures, arithmetic/logical operations, addressing modes and I/0. This application (which evolved over time) was to find the prime factorization of an integer. Specifically the application read an integer, found all prime divisors, stored them in an array, and then displayed the array The basic logic is given by the C++ code below.

```
void main()
{
   int A;        // number to factor
   int D[20];    // array of prime divisors
   cout << "Enter an integer > 1: ";
   cin >> A;
   int n = 2;     // initial trial divisor
   int last = 0; // number of items in D
   while (A > 1)
      {if ((A % n) == 0)// does n divide A?
         {D[last] = n;  // yes – store n
          last++;
          A = A / n;     // factor out n
         }
       else
          n++;    // no – increment divisor
      }
  for (int i = 0; i < last; i++)
     cout << D[i] << endl;
}
```

The operations needed to implement the above ran the gamut of programming techniques. The completed application

1. read and converted a string of ASCII digits to an integer
2. implemented integer division
3. used an indefinite loop to search for all prime divisors
4. inserted an item into an array
5. used a counting loop to access all items in an array
6. converted each integer to a string of ASCII digits and displayed each digit
7. consolidated one or more of the above into a subroutine with parameter passing

Although the prime factorization application could be fully implemented in each of the assemblers, this was not done. Given the differences in each assembler/architecture each piece of the application was implemented using the *most appropriate* assembler, the one that best illustrated a programming technique as constrained by the underlying architecture. For example, since both the Java Byte code simulator and the Intel 80x86 had a division instruction while the PDP-8 and ARC simulators didn't, it made more sense to do division in software only on the latter two. The effect was to do an incremental built of the prime factorization application over four architectures which cumulated with the completed application being done in Intel 80x86assembler. The four architectures/assemblers were covered sequentially starting with the simpler PDP-8 and ending with the Intel 80x86.

2. THE PDP-8

Of the four architectures, the PDP-8 was the simplest to learn but for large applications the most challenging to work with. It is (was) a 12 bit word machine with 12 bit addressing, fixed length instructions and only *eight* 3-bit op-codes. The PDP-8 had a single 12-bit Accumulator, an MQ register for multiplication/division and no index register(s). The lack of index registers was compensated for by indirect addressing. A carry out from the leftmost bit of the Accumulator was captured in a single bit Link register which gave the appearance of a combined Link/Accumulator register pair. Six of the op-codes were *memory reference instructions* (MRI); the two remaining op-codes used the bits of the address field as an op-code extension field for groups of instructions that manipulated the Link/ Accumulator and MQ registers. A number of these were conditional skip instructions which tested the accumulator and/or link bit. *Reversing* the test and following the condition skip with the unconditional MRI Jump (JMP) instruction resulted in a conditional branch operation.

The only arithmetic operation supported was a Two's complement Add (TAD) instruction. The non-MRI Complement and Increment Accumulator (CIA) instruction could be used to negate a number so subtraction was done by adding a negative value. Multiplication and division was done in software[1]

The MRI instruction Increment and Skip on Zero (ISZ) incremented the contents of its memory address skipping the next instruction if the result was zero. If the memory address was first initialized to a negative value, the skip on zero feature of ISZ paired with an unconditional jump instruction (JMP) implemented a counting loop. The remaining three MRI instructions were a Deposit and Clear Accumulator (DCA), a sort of destructive store, a Boolean AND instruction, and a Jump to Subroutine (JMS) instruction.

On the PDP-8 division for the prime factorization application was done by repeated subtraction. To compute $Q = A / B$ and $R = A\%B$ the divisor B was first negated and repeatedly added (i.e. subtracted) from the dividend A in the accumulator. As long as the result was positive, the quotient was incremented by the ISZ instruction.

```
    cla       // Clear Accumulator
    dca Q     // clear Q
    tad B     // load B (divisor)
    cia       // negate
    dca MB    // store at MB (minus divisor)
    tad A     // load A (dividend)
L2, tad MB    // subtract divisor
```

[1] The PDP-8 had an optional Extended Arithmetic Element (EAE) which did multiplication and division in hardware which was not implemented on the PDP-8simulator.

```
    spa       // Skip on Positive Accumulator
    jmp L3    // else exit loop
    isz Q     // increment Quotient
    jmp L2    // loop: note Q > 0
L3, tad B     // restore remainder
    dca R     // store AC to remainder
```

Since multiplication and division by *ten* is needed to read and write decimal integers, I/O was done in octal. Given a 12 bit unsigned octal integer, you left rotate the number three bits so the leading octal digit is in the lower three-bit of the accumulator, mask out the upper 9 bits, convert to ASCII by adding 48 decimal and print the ASCII character. The code given below does this using the ISZ-JMP configuration to loop 4 times. (The variables `m4`, `mask`, and `p48` contain the constants -4, 07(octal) and +48 respectively; the PDP-8 did not support immediate mode addressing.)

```
       cla cll       // clear AC and Link
       tad m4        // load -4
       dca cnt       // store in loop counter
L4,    cla cll       // clear AC and Link
       tad number    // load number
       cll ral       // left rotate
       szl
       iac
       cll ral       // left rotate
       szl
       iac
       cll ral       // left rotate
       szl
       iac
       dca number    // store it
       tad number    // load it
       and mask      // mask out upper 9 bits
       tad p48       // add 48 to convert to
                     // ASCII
       jms TYPE      // call the type
                     // subroutine to display
       isz cnt       // loop control
       jmp L4
```

The non-MRI Rotate Accumulator Left (RAL) instruction actually rotates the Link/Accumulator pair (a.k.a. rotate left with carry). To rotate the accumulator alone you had to CLear the Link bit (CLL) and then rotate left (when CLL and RAL instructions were combined the clear was executed before the rotate) then test the link bit using the Skip on Zero Link (SZL) instruction to skip over the Increment ACcumulator (IAC) instruction which would otherwise set the right most bit of the accumulator which was cleared by CLL RAL. A little reflection shows that if the left most bit of the accumulator was initially set or cleared, the right most bit of the accumulator was subsequently set or cleared.

Printing the ASCII digit in the accumulator was done by a subroutine call (JMS) of the TYPE routine which had been previously written.

At this point the strengths and weaknesses of the PDP-8 are apparent. While it is possible to accomplish anything with a very limited instruction set, it takes a lot of very clever and intricate coding to do so.

3. THE ARC – A RISC COMPUTER

In their textbook *Computer Architecture and Organization: An Integrated Approach* [2] authors Murcocca and Heuring introduce a very nice architecture/assembler called the ARC (A Risc Computer) which is based on the SPARC architecture. There is also a simulator which can be used to edit, assemble and execute simple ARC assembler programs.

The ARC is a 32-bit load and store architecture with thirty-two 32-bit registers (`%r0` through `%r31`) with register `%r0` being permanently wired to 0. Like the PDP-8, instructions are fixed length. Memory access is restricted to a register load instruction (`ld`) and a register store (`st`) instruction. Calculations are done in registers, all arithmetic and logical operations have three operands: two source and one destination. Thus a simple c = a + b instruction is implemented as

```
 ld [a], %r1          ! $r1 <- a
 ld [b], %r2          ! %r2 <- b
add %r1, %r2, %r3     ! %r3 = a + b
 st %r3, [c]          ! c <- %r3
```

Note the left to right data flow. Since register `%r0` is hardwired to 0, any result written to `%r0` is thrown away. Register `%r0` is also used to implement immediate mode addressing when registers can be initialized to hold small values (see below). Arithmetic and logical operations can either set or not set condition codes (`addcc` vs `add`; `subcc` vs `sub`). Conditional branching is done by testing the condition codes so for example `bl` will branch if the condition codes indicate a less than zero result. Unlike the PDP-8 there is no counting loop instruction (i.e. ISZ); instead a register containing the count is decremented until 0 is reached (see below).

The ARC simulator does not support a division (or multiplication) operation so as with the PDP-8 division is implemented in software. However, the richer instruction set of the ARC allows division to be implemented using the "standard" shift, test and restore algorithm (see Murdocca and Heurling [2] p. 71). In the code below register `%r1` holds the divisor, `%r2` holds the remainder, `%r3` holds the dividend/quotient and `%r4` is the loop counter (initialized to 32). A double register left shift shifts the

bits of the dividend into register %r2 where it is tested against the divisor. If subtraction of the divisor results in a positive value the least significant bit of %r3 (quotient) is set.

```
     xor   %r2,%r2,%r2 ! clear R
     add   %r0,32, %r4 ! init loop counter
d1:  addcc %r3,%r3,%r3 ! left shift
     add   %r2,%r2,%r2 !  %r2 - %r3 pair
     bcc   d2    ! branch on carry clear
     add   %r2,1  ,%r2 ! %r2 <- carry
d2:  subcc %r2,%r1,%r0 ! test if
     bl d3             !  %r1 ≤ %r2
     sub   %r2,%r1,%r2 ! if yes subtract
     add   %r3,1,  %r3 ! & set lsb of Q
d3:  subcc %r4,1,  %r4 ! decrement %r4
     bg d1             !  loop if > 0
```

It is instructive to study the nuances in the four lines of code needed to left shift the double register pair %r2-%r3. Left shifts of %r3 and %r2 are done using addcc and add respectively. If a carry into %r2 is needed, the branch on carry clear (bcc) instruction is not taken and 1 is added to %r2.

4. THE JAVA BYTE CODE SIMULATOR

Although the prime factorization application can be completely implemented on each architecture/simulator, the approach taken was to code those parts of the application which best exhibit some feature of a particular architecture. Because reading and writing decimal integers requires multiplication and division by ten which neither the PDP-8 nor the ARC simulators support in hardware, I/O of decimal integers not done. This is reserved for the Java Byte Code simulator which has built-in multiplication and division instructions.

The idea for a Java Byte Code simulator came from Tanenbaum's *Structured Computer Organization 4th Ed*. [6] where he introduces a micro-architecture simulator which implements a subset of Java Byte Code. Based on this, I wrote my own simulator which expanded the Java Byte code instruction set to include integer multiplication, division and modulo operations as well as including two non-standard I/O operations used by Tanenbaum: IN which pushes the next byte from the input buffer onto the stack and OUT which pops the byte on top of the stack to the output buffer (where it is subsequently displayed). This made the coding of decimal integer read and write routines fairly easy to do.

A stack architecture has no registers; instead the 32-bit operands are assumed to be on the stack and any results from an operation are automatically pushed back onto the stack. With implicit operands many instructions consist of a single byte length op-code making the instruction stream very compact; instructions are variable length ranging from one to three bytes.

References to memory are restricted to the push and pop operations iload and istore (the i is for integer). Hence writing assembler for the Java Byte Code engine has somewhat the feel of writing code for the load and store ARC. Since stack order is important there is a swap operation to reverse the two top items on the stack. Conditional branching is done by testing the top of the stack but since this pops the top of the stack, a dup (duplicate) operation is needed to make a copy first. A bipush (byte integer push) operation is used to push small integer constants onto the stack.

A good demonstration of the advantages of a stack architecture is seen in the way an unsigned decimal integer is displayed. The standard approach to output a decimal integer is to use repeated modulo 10 and divide by 10 operations to extract each digit. Given an integer you modulo by 10 to extract the digit then divide by 10 and go again with the result until zero is reached. For example 173%10 is 3 and 173/10 and 17. So after extracting the least significant digit 3 you continue with the 17. As this algorithm produces the digits in reverse order, a stack is needed to re-reverse them which is easily done with the stack oriented Java Byte Code architecture.

```
     iload number / push number on stack
     bipush 0     / push null to mark end
     swap         / swap null & number
P0:  dup          / duplicate TOS
     bipush 10    / push 10
     irem         / TOS = number % 10
     bipush 48    / push 48 = ASCII "0"
     iadd         / add to convert to ASCII
     swap         / bring number to top
     bipush 10    / push 10
     idiv         / TOS = number / 10
     dup          / duplicate TOS
     ifne P0      / if TOS ≠ 0 goto P0
     pop          / else done - throw out 0
P1:  dup          / duplicate TOS
     ifeq P2      / if TOS=0 goto p2 (done)
     out          / output TOS
     goto P1      / loop
P2:  pop          / done - pop null
```

5. OTHER PROGRAM TECHNIQUES

Lack of space permits only mentioning some of the other programming techniques which, due to architectural differences, were implemented various ways on the four architectures. Since the ARC and Intel 80x86 support indexed addressing, arrays were easiest to implement on these two (although they can be implemented on all four architectures). The same is true for subroutines calls and parameter passing; both were easier to do on the ARC and the Intel 80x86. The division algorithm on the ARC was coded as a subroutine using *pass by value in* for the divisor

and dividend and *pass by value out* for the quotient and remainder (as *opposed to pass by reference*). The mechanism of *pass by reference* for out parameters was covered when the prime factorization application was implemented on the Intel 80x86

6. THE INTEL 80x86

By the time the Intel 80x86 architecture was considered, all the pieces of the prime factorization application had been programmed on at least one of the three architectures. Implementing the application on the Intel 80x86 mostly required redoing the decimal integer read and write routines using a register architecture and implementing subroutine calls and parameter passing using pass by value and pass by reference parameters. Thus the final assignment ran: *In previous labs and assignments you wrote programs that found all the prime divisors of an integer. Write a similar program in Intel 80x86assembler which does the same. Your program will prompt for and a read a positive integer, find all the prime divisors and as each is found store it in an array, then print out the contents of the array. Your program should make efficient use of subroutines and parameter passing.*

7. SUMMARY

Assembler programming exercises are used to support and demonstrate the computer architectures being studied in a computer organization course. However as no one architecture/assembler embodies all the architectural features one would like to cover, I made use of four *different* architectures/assemblers. To partially offset the added time needed to learn the features (and idiosyncrasies) of each architecture/assembler a common application (or parts of it) was used for labs and assignments. Coding throughout course proceeded like an incremental build with pieces of the application being implemented on the architecture/assembler that best demonstrated some particular architectural feature. A common application reduced the burden of programming while at the same time provided a convenient way to compare and contrast the different architectures.

8. REFERENCES

[1] Levy, H. & Eckhouse, R. *Computer Programming and Architecture: The VAX* (2nd Ed). Digital Press. (1989).

[2] Murdocca, M & Heuring, V. *Computer Architecture and Organization: An Integrated Approach*. John Wiley & Sons. (2007).

[3] Null, Linda and Lobur, Julia. *The Essentials of Computer Organization and Architecture.* Jones and Bartlett. Sudbury MA. (2003)

[4] Shelburne, Brian J. (2003). Teaching Computer Organization using a PDP-8 Simulator *ACM/SIGCSE Bulletin: Thirty-Fourth SIGCSE Technical Symposium on Computer Science Education* (February 2003), 69-73

[5] Stallings, William. *Computer Organization & Architecture: Designing for Performance 6th Ed.* Prentice Hall, Upper Saddle River, NJ. (2003).

[6] Tanenbaum, Andrew A. *Structured Computer Organization 4th Ed*. Prentice Hall. Upper Saddle River, N.J. (1999).

[7] Warford, J. Stanley. *Computer Systems 2nd Ed.* Jones and Bartlett. Sudbury MA. (2002).

[8] Wolffe, Greg, Yurcik, William, Osborn, Hugh, Holliday, Mark, Teaching Computer Organization/ Architecture With Limits Resources Using Simulators. SIGCSE Bulletin 34 (March 2002). 176- 180

[9] *Special Issue on General Computer Architecture Simulators: ACM Journal of Educational Resources in Computing* (JERIC) Vol 1. No 4. December 2001.

[10] *Special Issue on General Computer Architecture Simulators: ACM Journal of Educational Resources in Computing (JERIC)* Vol 2. No 4. March 2002.