Looping and Use of Index Registers

This lecture discusses loop structures within assembly language,
and the language constructs evolved to support loops.

We begin with a review of type RX instructions, which are the
instructions that most naturally can use loop structures.

During these lectures, we shall follow a number of examples taken from a textbook
by Mr. George W. Struble of the University of Oregon.

Struble's textbook, published last in 1975, is out of print.

RX (Register–Indexed Storage) Format

This is a four–byte instruction of the form **OP R1,D2(X2,B2)**.

Type	Bytes		1	2	3	4
RX	4	R1,D2(X2,B2)	OP	R_1 X_2	B_2 D_2	D_2D_2

The first byte contains the 8–bit instruction code.

The second byte contains two 4–bit fields, each of which encodes a register number.

In order to illustrate this, consider the following data layout.

```
FW1  DC F'31'

     DC F'100'   Note that this full word is not labeled
```

Suppose that FW1 is at an address defined as offset **X'123'** from register 12.
As hexadecimal **C** is equal to decimal 12, the address would be specified as **C1 23**.

The next full word might have an address specified as **C1 27**, but we shall show another way to do the same thing. The code we shall consider is

```
L  R4,FW1      Load register 4 from the full word at FW1
AL R4,FW1+4    Add the value at the next full word address
```

RX (Register–Indexed Storage) Format (Continued)

This is a four–byte instruction of the form **`OP R1,D2(X2,B2)`**.

Type	Bytes		1	2	3	4
RX	4	R1,D2(X2,B2)	OP	R_1 X_2	B_2 D_2	D_2D_2

Consider the two line sequence of instructions

```
L  R4,FW1      Operation code is X'58'.

AL R4,FW1+4    Operation code is X'5E'.
```

The load instruction, remembering that the address of FW1 is specified as **`C1 23`**. The base register is R12, the displacement is **`X'123'`**, and there is no index register; so we have

```
58 40 C1 23
```

The next instruction is similar, except for its operation code.

```
5E 40 C1 27
```

RX Format (Using an Index Register)

Here we shall suppose that we want register 7 to be an index register.

As the second argument is at offset 4 from the first, we set R7 to have value 4.

This is a four–byte instruction of the form **`OP R1,D2(X2,B2)`**.

Type	Bytes		1	2	3	4
RX	4	R1,D2(X2,B2)	OP	R_1 X_2	B_2 D_2	D_2D_2

Consider the three line sequence of instructions

```
L  R7,=F‘4’    Register 7 gets the value 4.
L  R4,FW1      Operation code is X‘58’.
AL R4,FW1(R7)    Operation code is X‘5E’.
```

The object code for the last two instructions is now.

```
58 40 C1 23   This address is at displacement 123
              from the base address, which is in R12.

5E 47 C1 23   R7 contains the value 4.
              The address is at displacement 123 + 4
              or 127 from the base address, in R12.
```

Address Modification: Use of Index Registers

As noted above, a type RX instruction has the form **`OP R1,D2(X2,B2)`**.

This implies that the effective address is the sum of three values:

1. A displacement,
2. An address in a base register, and
3. A value in an index register.

Addresses may be modified by changing the values in any one of these three parts. The most natural of these choices is to change the value in the index register.

We now consider an example taken from a textbook Assembler Language Programming: the IBM System/360 and 370 (Second Edition) by George W. Struble.

The example concerns searching a list of numbers for a value that is specified in a given register. Struble uses R3, as we also shall do.

Before considering the entire loop, we should first examine a few lines of code as written in Mr. Struble's style. These can be quite interesting.

The Structure of an Indexed Address

Consider this line of code taken from the loop example.

```
LOOP     C   R3,ARG(R10)
```

The instruction is a Compare Fullword, which is a type RX instruction used
to compare the binary value in a register to that in a fullword at the indicated address.

As we shall see, the intent of this comparison is to set up for a **BE** instruction.

The item of real interest here is the second operand **ARG(R10)**.
How does the assembler map this to the form **D2(X2,B2)**?

> According to Struble (page 168) "The assembler inserts the address of an implied base register B2 for the second operand. The assembler also calculates and inserts the appropriate displacement D2 so that D2 and B2 together address ARG. The assembler also includes X2 = 10 [hexadecimal A] without knowledge or thought of the contents of register 10."

If register R12 (hexadecimal C) is being used as the implied base register, and if the label ARG is at displacement X'234' from the address in that register, the object code for the above instruction is **59 3A C2 34**.

Incrementing an Indexed Register

Much of this lecture will be focused on methods to change the value in the index register and so change the value of the effective address of the second argument.

This set of slides, which follows Struble's example closely, begins with an early example that he modifies to show the value of the really interesting instructions.

Consider the instruction **`LA R10,80(R10)`**. What does it do?

This instruction appears to be computing an address, but is it really doing that?

The **`LA`** instruction is indeed a "load address" instruction.

Consider the value that is to be loaded into register R10.

One takes the value already in R10, adds 80 to it, and places it back into R10.

In another style, this might be written as **`R10 = R10 + 80`**.

This line of code illustrates how programmers use features of the language.

Struble's First Loop

```
* PROGRAM TO SEARCH 20 NUMBERS AT ADDRESS ARG, ARG+80,
* ARG+160, ETC. FOR EQUALITY WITH A NUMBER IN REGISTER 3.
      LA   R10,0         SET VALUE IN R10 TO 0
LOOP  C    R3,ARG(R10)   COMPARE TO A NUMBER
      BE   OUT           IF FOUND, GO PROCESS IT.
      LA   R10,80(R10)   ADD 80 TO VALUE OF INDEX REGISTER.
      C    R10,=F'1600'  COMPARE TO 1600.
      BNE LOOP           IF NOT EQUAL, TRY AGAIN.
      DO SOMETHING HERE.  THE ARGUMENT IS NOT THERE
OUT   DO SOMETHING HERE
```

This program has only one obvious flaw, but it is a big one. The loop termination code should be **`BL LOOP`**.

In the example above, if the value goes from 1500 to 1700 and continues incrementing, the loop will never terminate properly.

Structure Analysis of Struble's First Loop

Here I analyze the structure that is implied by the program fragment. While I extend Struble's analysis, I remain entirely consistent with it.

```
START   Initialize the index register.

LOOP    Do the comparison and branch to OUT if equal

        Update the index register

        Test value in index register and loop if necessary.

FALL    Write the "fall through" code here.  The code
        immediately following the loop will execute only
        if the value is not found.

OUT     The value has been found.  The value in R10
        indicates its location in the data structure
        labeled as ARG.
```

Another Example from Struble

Here we have three zero–based arrays, each holding 20 fullword (32–bit) values.

We want an array sum, so that CC[K] = AA[K] + BB[K] for $0 \le K \le 19$.

Here is one way to do this, again using R10 as an index register.

```
         LA   R10,0          INITIALIZE THE INDEX REGISTER
LOOP     L    R4,AA(R10)     GET THE ELEMENT FROM ARRAY AA
         A    R4,BB(R10)     ADD THE ELEMENT FROM ARRAY BB
         ST   R4,CC(R10)     STORE THE ANSWER
         LA   R10,4(R10)     INCREMENT THE INDEX VALUE BY FOUR
         C    R10,=F‘80’     COMPARE TO 80
         BLT LOOP
```

Here we see a very important feature: the index register holds a byte offset for an address into an array and not an “index” in the sense of a high–level language.

How the VAX–11/780 Would Do This

The standard for the IBM System/360 and related mainframe computers is to use the index register as holding a **byte offset** from a base address.

In using this feature to move through an array of particular data types, the standard is to add the size of the data item to the index register.

More complex computers, such as the VAX–11/780 automatically account for the size of the data. Here is the above code written in the VAX style.

```
        LA   R10,0           INITIALIZE THE INDEX REGISTER
LOOP    L    R4,AA(R10)      GET THE ELEMENT FROM ARRAY AA
        A    R4,BB(R10)      ADD THE ELEMENT FROM ARRAY BB
        ST   R4,CC(R10++)    STORE THE ANSWER, INCREMENT INDEX
                             BY 1 FOR USE IN NEXT LOOP.
        C    R10,=F'20'      COMPARE TO 20
        BLT  LOOP
```

In the VAX style of programming, the address **`AA(R10)`** would be interpreted as **`AA + 4•(Value in R10)`**, because AA is an array of four–byte entries.

Other Options for the Loop

We now note a typical structure found in many loops. The loops tend to terminate with code of the form seen below.

```
LA REG,INCR(REG)    Increment the value
C REG,LIMIT         Compare to a limit value
BL LOOP             Branch if necessary
```

The only part of this structure that is not general is the assumption that the loop is "counting up". For a loop that counts down, we replace the last by **BH LOOP**.

A loop termination structure of this sort is so common that the architects of the IBM System/360 provided a number of special instructions to facilitate it.

The four instructions to be discussed here are as follows.

BXLE Branch on index lower or equal.

BXH Branch on index high.

BCT Branch on count. Easier to use, but less general than the above.

BCTR Branch on count to address in a register.

To Loop or Not To Loop

Consider the following code, which sums the contents of a table of a given size.
Here I assume that the table contents are 16–bit halfwords, beginning at address A0 and continuing at addresses A0+2, A0+4, etc.

The first code is the general loop. It is illustrated for an array of 50 two–byte entries.

```
         SR R6,R6        INITIALIZE INDEX REGISTER
         SR R7,R7        SET THE SUM TO 0
LOOP     AH R7,TAB(R6)   ADD ONE ELEMENT VALUE TO THE SUM
         A  R6,=F‘2’     INCREMENT TO NEXT HALFWORD ADDRESS
         C  R6,=F‘99’    LAST VALID OFFSET IS 98
         BL LOOP
```

On the other hand, if the table had only three entries, one might write the following code.

```
         LH R7,TAB
         AH R7,TAB+2
         AH R7,TAB+4
```

Branch on Index Value

The two instructions of interest here are:

BXLE Branch on index lower or equal. Op code = **X'87'.**

BXH Branch on index high. Op code = **X'86'.**

Each of these instructions is type RS; there are two register operands and a reference to a memory address. The form is `OP R1,R3,D2(B2)`.

Type	Bytes		1	2	3	4
RS	4	R1,R3,D2(B2)	OP	R_1 R_3	B_2 D_2	$D_2 D_2$

The first byte contains the 8–bit instruction code.

The second byte contains two 4–bit fields, each of which encodes a register number.

The first register is the one that will be incremented and then tested.

The third register indicates an even–odd register pair containing the increment value to be used and the limit value to which the incremented value is compared.

The third and fourth byte contain a 4–bit register number and 12–bit displacement, used to specify the memory address for the operand in storage.

The Even–Odd Register Pair

The form of each of the **BXLE** and **BHX** instructions is **OP R1,R3,D2(B2)**.

The source code form of the instructions might be **OP R1,R3,S2**, in which the argument S2 denotes the memory location with byte address indicated by **D2(B2)**.

The first register , **R1**, is the one that will be incremented and then tested.

The third register , **R3**, indicates the even register in an even–odd register pair. It is important to note that this value really should be an even number.

While the instruction can work if **R3** references an odd register, it tends to lead to the instruction showing bizarre unintended behavior.

The even register of the pair contains the increment to be applied to the register indicated by **R1**. It is important to note that the increment can be a negative number.

The odd register of the pair contains the limit to which the new value in the register indicated by **R1** will be compared.

NOTATION: **R3** will denote the even register of the pair, with contents **C(R3)**.

R3+1 will denote the odd register of the even–odd pair, with contents **C(R3+1)**.

Discussion of BXLE: Branch on Index Less Than or Equal

This could also be called “Branch on Index Not High”.

The instruction is written as **`BXLE R1,R3,S2`**.

The object code has the form **`87 R1,R3,D2(B2).`**

Step 1	Increment R1	**`R1 ← C(R1) + C(R3)`**
Step 2	Test the new value	**`Go to S2 if C(R1) ≤ C(R3 + 1)`**.

Assume that (R4) = 26, (R6) = 62, (R8) = 1, and (R9) = 40.

`BXLE 4,8,S2` The even–odd register pair is R8 and R9.

The value in R4 is incremented by the value in R8.

The value in R4 is now 27. This is compared to the value in R9. 27 ≤ 40, so the branch is taken.

`BXLE 6,8,S2` The even–odd register pair is R8 and R9.

The value in R6 is incremented by the value in R8.

The value in R6 is now 63. This is compared to the value in R9. 62 > 40, so the branch is not taken.

Discussion of BXH: Branch on Index High

The instruction is written as **`BXH R1,R3,S2`**.

The object code has the form **`86 R1,R3,D2(B2).`**

Step 1	Increment R1	**`R1 ← C(R1) + C(R3)`**
Step 2	Test the new value	**`Go to S2 if C(R1) > C(R3 + 1)`**.

Assume that (R4) = 4, (R6) = 12, (R8) = –4, and (R9) = 0.

`BXH 4,8,S2`

The even–odd register pair is R8 and R9.

The value in R4 is incremented by the value in R8.

The value in R4 is now 0. This is compared to the value in R9. $0 \le 0$, so the branch is not taken.

`BXH 6,8,S2`

The even–odd register pair is R8 and R9.

The value in R6 is incremented by the value in R8.

The value in R6 is now 8. This is compared to the value in R9. $8 > 0$, so the branch is taken.

A New Version of the Array Addition

Again we have three zero–based arrays, each holding 20 fullword (32–bit) values.

We want an array sum, so that CC[K] = AA[K] + BB[K] for $0 \le K \le 19$.

Here is one way to do this, again using R10 as an index register.

This time, we use BXLE with R8 as the increment register and R9 as the limit register.

```
       LA   R10,0            INITIALIZE THE INDEX REGISTER
       LA   R8,4             INCREMENT BY 4 BYTES FOR FULLWORD
       LA   R9,76            OFFSET OF 19TH ELEMENT
LOOP   L    R4,AA(R10)       GET THE ELEMENT FROM ARRAY AA
       A    R4,BB(R10)       ADD THE ELEMENT FROM ARRAY BB
       ST   R4,CC(R10)       STORE THE ANSWER
       BXLE R10,R8,LOOP      INCREMENT R10 BY 4, COMPARE TO 76
```

When the 19th element is processed R10 will have the value 76 (the proper byte offset).
After the 19th element is processed, R10 will be incremented to have the value 80,
and the branch will not be taken.

Polynomial Evaluation Using Horner's Rule

Horner's rule is a standard method for evaluating a polynomial for a given argument.

Let $P(X) = A_N \bullet X^N + A_{N-1} \bullet X^{N-1} + \ldots. + A_2 \bullet X^2 + A_1 \bullet X + A_0$.

Let X_0 be a specific value of the argument. Evaluate $P(X_0)$.

For example, let $P(X) = 2 \bullet X^3 + 5 \bullet X^2 - 7 \bullet X + 10$, with $X_0 = 2$.

Then $P(2) = 2 \bullet 8 + 5 \bullet 4 - 7 \bullet 2 + 10 = 16 + 20 - 14 + 10 = 32$.

Examination of the specific polynomial will show the motivation for Horner's rule.

$$
\begin{aligned}
P(X) &= 2 \bullet X^3 + 5 \bullet X^2 - 7 \bullet X + 10 \\
&= (2 \bullet X^2 + 5 \bullet X - 7) \bullet X + 10 \\
&= ([2 \bullet X + 5] \bullet X - 7) \bullet X + 10
\end{aligned}
$$

$$
\begin{aligned}
\text{So } P(2) &= ([2 \bullet 2 + 5] \bullet 2 - 7) \bullet 2 + 10 && = ([4 + 5] \bullet 2 - 7) \bullet 2 + 10 \\
&= ([9] \bullet 2 - 7) \bullet 2 + 10 && = (18 - 7) \bullet 2 + 10 \\
&= (11) \bullet 2 + 10 && = 22 + 10 = 32
\end{aligned}
$$

A Standard Algorithm for Horner's Rule

The basic loop is quite simple. In a higher–level language, we would something like the following, which has no error checking code.

```
P = A[N]
For J = (N – 1) Down To 0 Do
     P = P•X + A[J] ;
```

Consider again the polynomial $P(X) = 2\bullet X^3 + 5\bullet X^2 - 7\bullet X + 10$, with $X_0 = 2$.

In a notation appropriate for coding we have the following.
A[3] = 2, A[2] = 5, A[1] = – 7, A[0] = 10, and X0 = 2.

Let's use the loop above to evaluate the polynomial. N = 3.

Start with P = A[3] = 2.

J = 2	P = P•2 + A[2] = 2•2 + 5 = 9
J = 1	P = P•2 + A[1] = 9•2 – 7 = 11
J = 0	P = P•2 + A[0] = 11•2 + 10 = 32.

NOTE: This is entirely different from finding the root of a polynomial.

A Sketch of Our Algorithm for Horner's Rule

Our version of the assembler does not support explicit loops, so we write the equivalent code. In this, I shall use register names as "variables", so R3 will contain a value.

```
Algorithm Horner

On entry: R3 contains N, the degree of the polynomial
          R4 contains X, the value for evaluation.

      Set R8 = 0               This will be the answer.

      If R3 < 0 Go to END  No negative degrees

LOOP  R8 = R8•R4 + A0[R3]

      R3 = R3 - 1

      If R3 ≥ 0 Go to LOOP

END
```

This implementation will assume halfword arithmetic; all values are 16–bit integers.

More commonly, one would use floating–point arithmetic. Since my goal here is to illustrate the loop structure, I stick to the simpler arithmetic of halfwords.

More Notes on Our Implementation

The array will be laid out as a sequence of halfword (two byte) entries in memory.

The base address of the array will be denoted by the label A0.

There are (N + 1) entries, from A_0 through A_N, found at byte addresses A0, …, A0+2•N.
Here is an example for a 5th degree polynomial, with address offsets in decimal.

Address	A0	A0+2	A0+4	A0+6	A0+8	A0+10
Entry	A_0	A_1	A_2	A_3	A_4	A_5

Each halfword in the array will be referenced as **`A0(R3)`**, where R3 contains the byte offset of the item. The value in R3 will be set to 2•N and counted back to 0.

The increment value for **`R3`** is –2 (**`X'FFFFFFFE'`**), so that the register is actually decremented by 2. Its values are the byte offsets: 2•N, 2•N – 2, …, 4, 2, 0.

As I want to use the **`BXH`** instruction for this illustration, and want to allow for **`R3 = 0`**, I shall set the limit for the comparison to –1, though –2 would do as well.

At the last execution of the loop, R3 will be decremented from R3 = 0 to R3 = – 2.
The branch will not be taken.

Horner's Rule Polynomial Evaluation with BXH

```
*        ALGORITHM HORNER
*        ON ENTRY: R3 CONTAINS THE DEGREE OF THE POLYNOMIAL
*                  R4 CONTAINS THE VALUE OF X FOR P(X)
*        PROCESS:  R6 AND R7 WILL BE USED FOR THE BXH
*        ON EXIT:  R8 CONTAINS THE VALUE OF P(X).
         SR    R8,R8           SET R8 TO ZERO
         AR    R3,R3           DOUBLE R3 TO MAKE BYTE COUNT.
         LH    R6,=H'-2'       LOAD INCREMENT OF -2
         LH    R7,=H'-1'       LOAD LIMIT FOR TESTING
LOOP     MR    R8,R4           PRODUCT IN REGISTER PAIR R8,R9
                               FOR HALFWORDS, R9 IS NOT USED
         AH    R8,A0(R3)       ADD THE COEFFICIENT
         BXH   R3,R6,LOOP      LOOP IF C(R3) > -1.
```

For this example, I assume 16–bit integers (halfwords) for both the value of X and the values of all coefficients of the polynomial.

Given this, the sign bit in R8 will be correct after the multiplication and R9 is not used.

Branch on Count

The two instructions of interest now are:

BCT The branch on count instruction is a type RX instruction, with op code **X'46'**.

BCTR The branch on count (register) instruction is a type RR instruction.
This has op code **X'06'**.

The forms of the instructions are: **BCT R1,S2**

BCTR R1,R2.

Each of these instructions decrement the count in the R1 register by 1.

The actions of these instructions is described formally as follows.

BCT: **R1 ← C(R1) - 1**

Branch to S2 if C(R1) ≠ 0.

BCTR: **R1 ← C(R1) - 1**

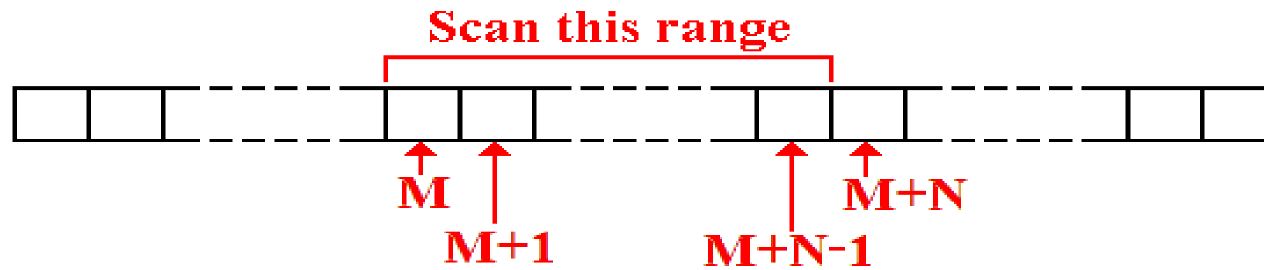
Branch to C(R2) if C(R1) ≠ 0 and R2 ≠ 0.

Note that **BCTR R1,0** will decrement R1 by 1, but not branch for any value in R1.

Scanning Text for Input/Output

Remember that input should be viewed as a card image of 80 columns and that output should be viewed as a line–printer line of 132 columns, with leading print control.

Consider a field of N characters found beginning in column M.



Suppose that the leftmost byte in this array is associated with the label **BASE**.

The leftmost byte in the range of interest will be denoted by the label **BASE+M**.

Elements in this range will be referenced using an index register as **BASE+M(Reg)**.

For example, suppose that the field of interest contains 12 characters and begins with column 20. It then goes between columns 20 and 31, inclusive.

Using R3 as an index, we reference this as **BASE+20(R3)**, with $0 \leq (R3) < 20$.

Scanning left to right will use BXLE and scanning right to left will use BXH.