Base Registers

Today, we cover the use of base registers in address calculation.

Topics for today include.

1. A review of 32–bit binary arithmetic, as described by IBM.

2. A characterization of the sixteen general–purpose registers in the System/370. Which are really general purpose?

3. Control sections and their relation to address calculation.


5. Assigning and loading base registers.

6. Instruction formats and the use of base registers.
32–bit binary arithmetic, as described by IBM.

The System/370 uses 32–bit two’s–complement signed integers.
The range of these integers is \(-2^{31}\) to \(2^{31} – 1\), inclusive.
Now, \(2^{31} = 2,147,483,648\), so the range is \(-2,147,483,648\) through \(2,147,483,6647\).
Each of the sixteen general–purpose registers contains binary integer data.
The bits in the registers are numbered left to right. Notice that this is not the standard used in other courses.
Bit 0 is the sign bit. It is 1 for a negative number and 0 for a non–negative.
In the book’s terminology, “bits 1–31 are data”. This is not the way I would say it, but it is the terminology we shall use for this course.

Addressing
The general purpose registers perform all the addressing involved in referencing main storage. When used for addressing, the register is called a base register.
When so used, the contents of the register are viewed as an unsigned integer.
Some System/370 systems provide only 24 bits for addressing.
Current System/390 systems use 64–bit addressing.
The General Purpose Registers

The general–purpose registers in the System/370 are identified by number: 0 – 15. Of these, only the ten registers 3 through 12 can be used for any purpose. The other six registers are “less general purpose” and should be used with caution.

**Registers 0 and 1** can be used as temporary registers, but calls to supervisor routines will destroy their contents.

**Register 2** can be used as a temporary and possibly as a base register. The TRT (Translate and Test) instruction will change the value of this register.

**Registers 13, 14, and 15** are used by the control programs and subprograms.

Each of the sixteen registers is identified by a four–bit binary number, or equivalently by a single hexadecimal digit.

Suggested convention: Use either **register 3** or **register 12** as the single required base register. The standard prefix code should contain the following sequence.

```
BALR 12, 0     This uses register 12 as a base.
USING *, 12
```
Register Naming Conventions

Note that there are two ways of writing the part of the standard prefix code.

What we have discussed in these slides:

```
BALR  12, 0
USING *, 12
```

What we have used in our labs is valid if it follows a “R12 EQU 12”:

```
BALR  R12, 0
USING *, R12
```

From this I infer that the assembler will accept either notation in places in which the syntax expects a register to be used.

NOTE: The first line is a truncated subroutine call.

The instruction says:

1. Store the address of the next instruction into R12
2. Branch to the address in the second argument.

Since that second address is 0, the branch is not taken and execution continues.
CSECT: Control Sections

By definition, a control section (CSECT) is “a block of coding that can be relocated (independent of other coding) without altering the operating logic of the program”.

For the modern programmer, a CSECT might be considered as an independent code module or package. The match in terminology is not precise, but suggestive.

Very large assembly language programs will require more than one control section. [My note: This is probably old information.]

In any case, the programs we write will require only one CSECT, even assuming the addressing constraints imposed by the old System/370 architecture.

A program with a single CSECT may be started in one of two ways. For example, the program LAB01 may be started in one of two ways.

```
LAB01 START
LAB01 CSECT
```

The assembler allows for the specification of a load address in the START or CSECT. This is probably a bad idea; it may be ignored by a modern operating system.

The last line in the CSECT must be **END** Name, such as **END  LAB01**.
Base Register Addressing

The System/370 uses a common design feature that splits addresses into two parts:

1. A base address, stored in a specified base register. In general, only registers 3 through 12 should be used as base registers.

2. A displacement, specifying the positive offset (in bytes) from the start of the section. The System/370 uses a 12–bit number for this displacement. The displacement value is in the range 0 through 4095, inclusive.

The format of the address in this form is as follows:

\[
| B | D D D |
\]

where B is the single hexadecimal digit indicating the base register, and “D D D” denotes the three hexadecimal digits used to specify the offset.

Suppose that general–purpose register 3 contains the value X’4500’.

The address reference 3507, shown as | 3 | 507 | refers to the address X’4500’ + X’507’ = X’4A07’ in hexadecimal 5 + 5 = A.

NOTE: Register 0 cannot be used as a base register. The assembler will interpret the address | 0 | D D D | as no base register being used.
More on Register 0 As a Base Register

The bottom line is “Don’t try this (at home)”. In other words, any attempt to use register 0 for anything other than temporary results is likely to cause problems.

As noted above, some addresses are given in the form | B | D D D |, where

- B is a hexadecimal digit indicating a base register, and

- D D D is a set of three hexadecimal digits indicating an offset in the range 0 – 4095.

If the object code generated has the form | 0 | D D D |, then no base register is used in computing the address. We shall use this later to load registers with positive constants.

One has another option that might force register 0 to be a base register. We can start the program as follows.

```
BALR R0, 0
USING *, R0
```

While this MIGHT assemble correctly (I have no idea), it is most certainly a very bad idea. Any call to a system procedure will disrupt the addressing.
Options: No Base Register vs. The Default Base Register

So far, we have considered only the object code form of a typical address. We now “jump ahead” a bit and look at two typical instructions that use this address type.

One type of instruction, called “RS”, used for register–to–storage instructions. Such an instruction has source code of the form \( \text{OP R1,R3,D2 (B2)} \).

Such an instruction has object code of the form \( \text{OP R_1R_3 B_2D_2 D_2D_2} \).

We look at \text{LM}, an interesting example of this format.

\text{LM R1,R3,S2} loads multiple registers in the range R1 – R3 from the memory location specified by \text{S2}, the address of which will be in the form \(| \text{B_2 | D_2 D_2 D_2} |\).

We now interpret the following code fragment.

\begin{align*}
\text{BALR R12, 0} & \quad \text{Establish register R12 (X'C')}
\text{USING *, R12} & \quad \text{as the default base register.}
\text{LM R5,R7,S2} & \quad \text{might have object code 98 57 C1 00.}
& \quad \text{This uses the default base register.}
\text{LM R9,R11,S3(R3)} & \quad \text{might have object code 98 9B 32 00.}
\end{align*}

Object code such as \text{98 9B 0E 00} would call for use of an absolute address, not computed from a base register. For this example, it is likely to be bad code.
Rationale for Base Register Addressing

The textbook offers two advantages of base/displacement addressing. One reason is still valid and one shows an interesting history.

Remember that the System/370 of the time admitted a 24–bit address space, with addresses ranging from 0 through $2^{24} - 1$ or 0 through 16,777,215.

A full 24–bit address would require 24 bits, or six hexadecimal digits, or three bytes.

The base register/displacement method of addressing allocates

- 4 bits to the base register
- 12 bits to the displacement

In this method, an address requires 16 bits, or two bytes.

We now quote Abel on the first major advantage of base/displacement addressing

“The instruction length is reduced because each address requires only two bytes rather than three.”

One might infer that some of the System/360 and System/370 installations had very little memory.
Base Register/Displacement Addressing: Relocating the Code

The second major advantage of base/displacement addressing still applies today.

“The system facilitates program relocatability. Instead of assigning specific [fixed] storage addresses, the assembler determines each address relative to a base address. At execute time [after the program is loaded], the base address, which may be anywhere in storage, is loaded into a base register.”

The standard prefix code

```
BALR 3, 0
USING *, 3     Not an instruction; no address assigned
```

may be translated as follows:

1. What is the address of the first instruction after the “USING”? 
2. Load that address into register 3 and use it as an base address in that register.

The other option for relocating code is to use a relocating loader to adjust fixed address references to reflect the starting address of the code. This is also acceptable.

In the 1960’s, code that did not reference absolute addresses was thought to be superior. It was called “position independent code”.
Base/Displacement vs. Indexed Addressing

Note the similarities with indexed addressing, in which the base is given by a variable and the offset is given by a register. Systems that use the register contents as a base do so because the registers can store larger numbers than the bits in the machine code.

For example, the System/360 allocates only 12 bits for the displacement, which is combined with the contents of a register, which can be a 32–bit number.

The MIPS–32, a design from the mid 1980’s, also uses base/displacement addressing rather than indexed addressing.

In the MIPS–32 architecture, the displacement is a 16–bit signed integer, and the register values are 32–bit numbers.

This is basically the same idea, except that the displacement can be a negative number. The range is –32,768 through 32,767 inclusive. As the addresses in the MIPS–32 must be multiples of four, this displacement is converted to an address offset by first multiplying it by four. The effective offset values are multiples of four, ranging from –131,072 through 131,068.
Addressing: More Discussion

Here are some more examples of addressing using an index register and a base register. All of these examples are taken from type RX instructions, which use indexing.

Each of these is a four-byte instruction of the form \( OP \, R_1, D_2 (X_2, B_2) \). The format of the object code is \( OP \, R_1X_2 \, B_2D_2 \, D_2D_2 \). Each byte contains two hexadecimal digits.

We interpret the 32-bit object code as follows.

- \( OP \) This is an eight-bit operation code.
- \( R_1X_2 \) This byte contains two hexadecimal digits, each of which is significant.
  - \( R_1 \) denotes a register as the source or destination of the operation.
  - \( X_2 \) denotes a general-purpose register to be used as an index register.
- \( B_2D_2 \, D_2D_2 \) This contains the argument address as a base register and displacement.

Remember that the displacement, given by three hexadecimal digits, is treated as a 12-bit unsigned integer. In decimal, the limit is \( 0 \leq \text{Displacement} \leq 4095 \).

The general form by which an address is computed is

\[
\text{Contents (Base Register)} + \text{Contents (Index Register)} + \text{Displacement}
\]

Some instructions do not use index register addressing.
Addressing: Example 1

Here is some object code for analysis.

\[ 58 \ 40 \ C1 \ 23 \]

The first thing to note is that the opcode, \( 58 \), is that for \( L \), a Register Load. This is a type RX instruction with object code of the form \( \text{OP} \ R_1X_2 \ B_2D_2 \ D_2D_2 \).

As noted above, \( \text{OP} = 58 \).

We see that \( R_1 = 4 \). It is register 4 that is being loaded from memory.

We see that \( X_2 = 0 \). Indexed addressing is not used.

We also note that \( B_2D_2 \ D_2D_2 = C1 \ 23 \), indicating an offset of \( \text{X}'123' \) from the address value stored in general–purpose register 12 (hexadecimal \( C \)).

Suppose that the value in general–purpose register 12 is \( \text{X}'2500' \). The effective address for this instruction is then \( \text{X}'2500' + \text{X}'123' = \text{X}'2623' \).
Addressing: Example 2

Here is another example of object code.

\[ \text{58 A7 B1 25} \]

The first thing to note is that the opcode, 58, is that for \text{L}, a Register Load. This is a type RX instruction with object code of the form \text{OP R1X B2D D2D}.

As noted above, \text{OP} = 58.

The hexadecimal digit for the register is A, indicating that register 10 is being loaded. Recall that all of the digits in the object code are given in hexadecimal.

We see that \text{X2} = 7, indicating that general–purpose register 7 is being used as an index register.

We also note that \text{B2D D2D} = \text{B1 25}, indicating an offset of \text{X'd25'} from the address value stored in general–purpose register 11 (hexadecimal B).

Suppose the following:  
Register 11 contains \text{X'0012 4000'}
The displacement is \text{X'0000 0125'}
Register 7 contains \text{X'0000 0300'}

The address is thus \text{X'0012 4425'}. 
An Aside: When Is It NOT An Address?

Let’s look at the standard form used for a base & displacement address.

| B | D D D |

Technically, the 12–bit unsigned integer indicated by D D D is added to the contents of the register indicated by B, and the results used as an address.

There are instructions in which the value so computed is just used as a value and not as an address. Consider the instruction SLL R4,1.

It is assembled as shown below.

000018 8940 0001 00001 48   SLL R4,1

The base register is 0, indicating that no base register is used. The “offset” is 001. The value by which to shift is given as the sum, which is 1.

We could use a standard register to hold the value of the shift count, as in the following, which uses R8 to hold the shift count.

000018 8940 8000 00001 48   SLL R4,0 (R8)

**NOTE:** This is a good example of not using a “base register” in order to generate an “absolute constant”, not relative to any address. Here, the value is a count.
Assigning and loading base registers.

If the program is to use a base register for base register/displacement addressing, that register must be specified and provided with an initial value.

Again, the standard prefix code handles this.

```
BALR 12, 0
USING *, 12
```

If register 12 is used as a base register, it cannot be used for any other purpose.

In other words, your code should not reference register 12 explicitly.

We have two standards suggested for a base register. The textbook uses register 3 and one of our examples uses register 12. Pick one and use it consistently.
The Standard OS Prefix Code

Just to be complete, we show typical prefix code for running under OS.

This is taken from our lab 1.

LAB1    CSECT ,         COMMA REQUIRED IF COMMENT ON THIS STMT
************************************************************** *
* STANDARD LINKAGE FOR A REUSABLE OS/MVS CSECT
************************************************************** *

```
SAVE   (14,12)       SAVE CALLER'S REGS
BALR   R12,0         ESTABLISH
USING  *,R12         ADDRESSABILITY
LA     R2,SAVEAREA   POINT TO MY LOWER-LEVEL SA
ST     R2,8 (,R13)   FORWARD-CHAIN MINE FROM CALLER'S
ST     R13,SAVEAREA+4 BACK-CHAIN CALLER'S FROM MINE
LR     R13,R2        SET 13 FOR MY SUBROUTINE CALLS
```

NOTE that this code uses register 12 as the base register.
**Instruction Formats**

There are six instruction formats of interest to us. These are classified by the type of transfer.

- **RR**  Register to register transfers.
- **RS**  Register to storage and register from storage
- **RX**  Register to indexed storage and register from indexed storage
- **SI**  Storage immediate
- **SS**  There are two variants of these Storage to storage transfer instructions.