CHAPTER THREE

Operands

Since the appearance of the early architectures, many changes have been made to improve computer performance. In general, however, the basic architecture remains. This chapter addresses the improvements made to computer architecture with respect to addressing modes and how they affect the performance of the processor.

3.1 Addressing Modes

Now that we have an idea of the general operation of the early stored program machines, let us see how they can be improved first by addressing the kinds of data or operands that an instruction might need.

The instruction set of the Princeton IAS machine used at most a single operand pointing to a datum stored in memory. For example, there were eight commands that used the add circuitry to either load, add, or subtract the value, absolute value, or negated absolute value of a value from memory. (The load commands used the add circuitry by first clearing the accumulator, then performing an add.) For all of these instructions, there is only one data source that can be added to the accumulator, i.e., a value that comes from an address in memory.

This makes some basic operations inefficient if not difficult. For example, adding a constant to the accumulator using the IAS machine involved initializing the constant in memory before runtime, then referencing that memory location during execution to get the constant.

A simpler method would be to create a new opcode that treated the instruction's operand as a constant rather than a memory address. If the original IAS machine instruction format was maintained, this would limit constants to 12-bits, but it would simplify the program by removing the need for a region of memory to be reserved for constants. In addition, instruction execution would be faster because only the instruction would need to be read from memory. There would no longer be a need for a second memory read for the constant.

Allowing for instructions that use constants instead of addresses would force the instruction set designer to reserve additional machine codes for these new operations. For example, if we wanted to allow for a constant operand to be added, subtracted, multiplied, or divided with

the accumulator, then four additional machine code values would need to be reserved for these instructions.

The IAS machine's computational instructions also "implied" a second data source, i.e., the accumulator. Since the machine had only one accumulator, every computational operation assumed the accumulator was the second data source *and* the result destination.

Accumulators are vital. They allow intermediate values to remain in the computational unit instead of being stored in memory thereby requiring additional memory operations. For example, computing the expression $A \times (B + C)$ requires two steps: add B and C, then multiply by A. The intermediate result, $(B + C)$, must be stored somewhere. If that somewhere is memory, an additional store operation must occur. An accumulator can hold the intermediate result until a final value has been computed. $A \times (B + C)$ would then become, "load B to accumulator, add C to accumulator, and multiply A with accumulator."

What if, however, there were two or more accumulators? Additional accumulators would require the instruction not only to identify the operation, but also the accumulator(s) on which to operate. In fact, there may be additional operations that use more than one accumulator, e.g., adding two accumulators together or using two accumulators as a single "double-wide" operand. Just like adding the option of using a constant as an operand, additional accumulators would require additional machine codes to be reserved. If this was applied only to computational instructions such as add, subtract, multiply, and divide, the addition of a second accumulator would add another four opcodes to our instruction set.

The use of constants or multiple accumulators allows us to program more efficiently, but it does not present more programming options over the basic von Neumann architecture. Greater programming flexibility can be found through the use of pointers.

A pointer is a data element that is not a computational value, rather it points to a value in memory by storing the value's address. This address could be used to reference and step through a string or array in memory or it could be used to allow a single section of code to operate on any memory location by simply changing the value of a pointer.

There are two places where pointers can be stored: a special register or a memory location. If a special register is used, the register must be the same size as a memory address. The IAS machine used two pointer registers: MAR and PC. The functions of these two registers were

fixed, however, so the programmer could not take advantage of them as general purpose pointers.

Assume we create a 10-bit register called *memory pointer* or *MP*. A special "load pointer" instruction would be required that would initialize the pointer. Additional instructions could then be included to use MP as an operand source.

For example, let's assume a programmer needed to write a section of code to count the number of times a certain value appeared in an array. To begin with, MP could be loaded with the starting address of the array to be searched. The accumulator could then be loaded with the value from memory that is pointed to by MP, compared with the search value, and if it is equal, increment and store a counter. To go to the next element of the array, MP would be incremented. If MP doesn't yet point to the end of the array, the accumulator could be loaded with the next value in order to do the compare again.

This type of search could have been done with the IAS machine in its original form, but it would not have been efficient. Two instructions were included in the original IAS description that allowed for self-modification of the code. Remember that each memory location held a 40-bit value or two 20-bit instructions. The last two opcodes of Table 2-4 were used to replace the address portion of an instruction with the ten least significant bits of the accumulator. One of the opcodes was used for the left instruction while the other modified the right instruction. Using these opcodes, the address pointed to by the instruction could be modified by directly modifying the instruction.

As with the other addressing modes, adding a pointer register such as MP requires additional opcodes to modify and use it. These opcodes would include instructions to load and store the register, increment and decrement it, and possibly add to or subtract from it. Therefore, we would have to add up to six instructions to our instruction set for the processor to access the pointer register, not to mention the instructions required to use it as a data source.

A pointer can also be stored in memory rather than in a special processor register. This would allow the number of pointers used by a program to be limited only by the size of memory. It would also make any existing memory-based opcodes available to the pointers. There is a problem with this beyond increasing the size of the instruction set. Using memory to contain a pointer adds at least one memory access to

any pointer-based instruction: read instruction from memory, read pointer from memory, and use pointer to read data from memory.

Single-dimensional arrays and strings are easy to process using pointers. When it comes to a complex data structure containing multiple data types arranged at different offsets from a base address, it may be necessary to once again increase our addressing capabilities.

Data structures bring about the need for an additional method for addressing called ***offset addressing***. Offset addressing requires two components: a base address and an offset. The base address points to the beginning of a complex data structure while the offset identifies the number of memory locations away from the base address where the data of interest is located. Typically, the base address is contained in one of the processor's general purpose pointer registers. The offset could be a constant that is included as part of the instruction's operand. For example, if our MP register contained the base address 0x1000 and the offset as defined by an instruction's operand were 0x1a, then the physical address that the instruction would get its data from would be 0x1000 + 0x1a = 0x101a.

This method can be expanded to use a third component, an index, to calculate an operand's address. In this case there would be three address components - a base address to be added to an index value which is then added to an offset. There are also multiple options for the source of the offset. The offset may be a constant operand, it may be in a register, or it may be indirectly referenced using a pointer. Once again, the more addressing modes that a processor allows, the more flexible the instruction set is. This comes at the cost of a more complex the instruction set.

A common use for general purpose pointers is to process an array or string by stepping through its memory addresses. After one address has been processed, the pointer is either incremented or decremented to the next array or string element. Many processors include opcodes that first use the general purpose pointer, then automatically increment or decrement it after the instruction has been completed. This is called auto-indexing. Some processors use a configurable flag to determine whether a pointer is automatically incremented or decremented at the completion of the operation.

One common form of auto-indexing is the program counter or PC. This special purpose pointer points to the next instruction to execute and is automatically incremented to the next instruction once the

current instruction has been fetched. There are some cases where PC is modified by code, specifically during a branch or jump instruction. An unconditional jump typically loads the PC with a new value, i.e., the address to be jumped to. In some cases, however, a signed offset is added to PC allowing for both forward and backward jumps in code. This is called relative addressing.

In general, a processor's addressing capabilities must allow access to the entire memory range while still allowing for enough flexibility to handle multiple programming needs. It must be well-defined enough that the address can be fixed at assemble time or computed during run time. The following is a summary of the types addressing modes that might be available in a processor's instruction set.

- ***Implied addressing:*** No address is used. For example, commands halting the processor or forcing an interrupt do not use data.
- ***Immediate addressing:*** The instruction's operand is the data, i.e., the data is contained within the instruction. This would be how constants are handled. The data is fixed at assemble time.
- ***Register addressing:*** The instruction refers to a processor register where it will find the required data. Register selection is fixed at assemble time.
- ***Direct addressing:*** The operand field of the instruction is an address. Address is fixed at assemble time, but the value at that address can change during program execution.
- ***Register indirect addressing:*** The instruction refers to a processor register where it will find the address in memory where the required data is stored. Register selection is fixed at assemble time, but the address within the register is set at run time.
- ***Indirect addressing:*** The operand field of the instruction specifies an address which contains the address of the data. This requires two memory accesses: the first to fetch the pointer and the second to fetch the operand.
- ***Indexed addressing:*** The instruction has two operand references: a reference to a processor register where it will find a base address in memory and a second reference, either a constant or a second register, with the offset from the base address where the required data will be found. This addressing mode may also be combined with auto-indexing. Register selection is fixed at assemble time.

- ***Base + indirect offset addressing:*** The instruction has two operand references both of which are addresses. The first address points to the base address and the second address points to an indirect reference to the offset from the base address. The addresses containing the base address and the offset address are fixed at assemble time. The addresses they contain are set at run time.
- ***Base + offset + indirect offset addressing:*** This is a combination of indexed and base + indirect offset addressing where both an indirect offset and a constant or register value is added to the base address.
- ***Relative addressing:*** The operand field of the instruction contains a signed offset to be added to the program counter. This offset is set at assemble time.
- ***Stack addressing:*** Much like PC, the stack pointer (SP) has unique addressing requirements. The execution of a push or pop instruction performs an auto-index of SP. Depending on the processor there are four possible cases for stack addressing. The stack pointer may be auto-incremented or auto-decremented either before or after execution of the push or pop.

3.2 Number of Operands in an Instruction

Now that we understand instruction data sources or operands, we should turn our attention to how they are used. The typical instruction combines two data sources in order to generate a third result. For example, an add instruction might look like $A = B + C$. In the case of the IAS machine, one of the data sources for the add operation is pointed to by the 10-bit memory address contained in the last ten bits of the instruction while the other data source is the accumulator. For the von Neumann architecture, the accumulator is implied, but for some processors, both of the data sources might be identified explicitly. If this is the case, the instruction format must allow for multiple data sources to be named.

With the IAS machine, the destination of the result is also implied - it is always the accumulator. This is not the case with all processors. Some processors allow the instruction to also specify the destination of the result.

Assume that we need perform the operation $A = B + C$ where A, B, and C all refer to memory addresses. Depending on the available instruction formats, this simple operation could take one of four possible forms.

At the far end of the spectrum is the ***3-address instruction***. This is the most complex instruction format allowing a single instruction to represent all three data addresses: the two data sources and the result destination. In this case, the code needed to realize $A = B + C$ would consist of a single instruction:

```
ADD A,B,C
```

Processors utilizing the 3-address instruction format treat memory as a large set of registers. Typically this is not desired since memory accesses are one of the main bottlenecks of processor performance. In addition, the ability to refer to three addresses with a single instruction tends to increase the instruction set's complexity. This format, however, does allow for a cleaner translation between high-level programming languages and the assembly language code that realizes it.

It is possible to add a ***fourth address*** to the 3-address instruction, but this occurs in only rare, special cases. The fourth address is typically used for program flow-control. In other words, the outcome of the operation might force a branch to the address specified by the fourth operand in the instruction.

The next type of instruction is the ***2-address instruction***. In this case, one of the operands plays the part of both a data source and the result destination. This format mimics the C-style combined assignment operators, e.g., $A \mathrel{+}= B$.

This adds a step to the assembly language realization of our addition example by forcing us to load A with B before adding C to it. This instruction format is far more common than the 3-address instruction, but since A is now written to twice, we've also added an additional memory write.

```
COPY A,B
ADD A,C
```

The IAS machine utilizes a ***1-address instruction***. To do this, one of the operands must be implied, specifically the one that acts both as a data source and as the result destination. The function operates as $Acc = Acc + B$ where *Acc* represents a processor accumulator. It is for this reason that 1-address instructions are typically referred to as ***accumulator-based operations***.

Going to a 1-address instruction makes our code a little more complicated. The sample code below assumes that the processor has a single accumulator. The load instruction loads a value from a specified memory address into the accumulator and the store instruction stores the accumulator to a specified memory address.

```
LOAD B
ADD C
STORE A
```

Last of all, there are ***0-address instruction*** machines, typically referred to as ***stack machines***. In place of registers, these machines use the stack. Push instructions are used to populate the stack with data. To execute an instruction, the top one or two values are popped from the stack, are operated on, and the result is pushed back to the stack.

To perform the simple addition, the two numbers, B and C, are placed on the top of the stack. Once they are on the stack, a 0-address "add" is performed which removes the numbers from the stack, adds them together, then places the result back onto the stack. A final pop can be used to store the result to memory.

```
PUSH B
PUSH C
ADD
POP A
```

Before the advent of the parenthesis key on calculators, most calculators could only perform a single operation between a number stored in the accumulator and a new number entered on the keypad. To get around this limitation, Hewlett-Packard began successfully marketing calculators during the 1960's that used a mathematical notation referred to as postfix or reverse-Polish notation (RPN). The notation was named in honor of Jan Łukasiewicz, the developer of both RPN and prefix or Polish notation.

RPN represents expressions by first listing the two operands, then identifying the operation. For example, $B + C$ in RPN is $B\ C\ +$. More complex expressions such as $D \times (B + C)$ could be represented in this notation too: $D\ B\ C\ +\ \times$. For those who are unfamiliar with the notation, it may seem quite odd, but for the people at HP, it was a

breakthrough in being able to represent complex expressions without the use of parenthesis.

To evaluate an RPN expression, search the string from left to right for a substring of the form *XYf* where *X* and *Y* are values and *f* is a symbol for a two-input operation. Evaluate *XYf* and replace the substring with the result. Repeat this operation until you reach a single result. If you are unable to reach a single result, then the expression was not in proper RPN format.

What does RPN have to do with 0-address instructions? Computers in general adapt well to RPN. Just take a look for example at the process of adding two numbers. Load the first number into register A, load the second number into register B, execute an ADD A,B instruction, i.e., load number, load number, execute function.

Exercise

Evaluate the RPN expression 2 1 – 3 4 2 ÷ + ×

Solution

The steps below represent each time the pattern *XYf* is found and replaced with a result.

2 1 – 3 4 2 ÷ + ×	1st pattern: 2 1 – is replaced with 2 – 1 = 1
1 3 4 2 ÷ + ×	2nd pattern: 4 2 ÷ is replaced with 4 ÷ 2 = 2
1 3 2 + ×	3rd pattern: 3 2 + is replaced with 3 + 2 = 5
1 5 ×	4th pattern: 1 5 × is replaced with 1 × 5 = 5
5	

The evaluated expression is (2 – 1) × (3 + (4 ÷ 2)) = 5.

The relation between RPN and processors is most obvious when discussing a stack machine. Look at the expression from the previous exercise. If we treat each value as a push operation and each mathematical operator as a command, we convert 2 1 – 3 4 2 ÷ + × to the following sequence of instructions.

```
PUSH 2
PUSH 1
SUB
PUSH 3
PUSH 4
PUSH 2
DIV
ADD
MULT
POP A
```

The first two instructions push 2 and 1 onto the stack. The SUB instruction then subtracts 1 from 2 and puts the result of 1 back on the stack. Now a 1 is on the top of the stack.

The next four instructions push 3, 4, and 2 onto the stack above stored 1. The top two values are now 2 and 4, so when the DIV instruction is executed, 4 is divided by 2 and the result of 2 is placed back on the stack. This means that the stack now contains, in order from newest to oldest, 2, 3, and 1.

Next, the ADD instruction is executed taking the 2 and the 3, adding them together to get 5, and then placing the 5 on top of the stack. Now the stack contains 5 and 1.

The MULT instruction multiplies 5 and 1 and puts the result of 5 back on the stack. The POP A pulls off the 5 and stores it in memory location A.

But how do you take a mathematical expression and convert it to RPN? Dutch-born computer scientist Edsger Dijkstra developed a process known as the "shunting yard algorithm" which takes an infix (the form of mathematical expressions that most people are familiar with) and converts it to postfix or RPN. The term "shunting yard" comes from the fact that the process mimics the operation of a rail yard where the values, operations, and symbols of a mathematical expression are rearranged in the same manner as railcars would be.

To perform the shunting yard algorithm, the mathematical expression is treated as a string, each element of which is routed either to an output expression or a stack depending on its type (value, operation, or parenthesis). Once all of the elements of the expression have been processed, the final output expression should be in RPN notation. If there is an error in the format of the mathematical

expression such as mismatched parentheses, there will also be an error in the RPN notation.

The term stack is used here to represent a last in, first-out (LIFO) method for keeping track of operators and parentheses appearing in the expression. If a sequence of elements is to be retrieved from the stack and sent to the output, they are to be sent one at a time and in the reverse order from which they were added to the stack.

The output is generated by adding the reordered elements of the original expression one at a time by appending them to the right side of the RPN expression.

Begin by evaluating the expression one element at a time from left to right. Each element will either be a value, an operation, or a parenthesis. A value is sent directly to the output. A left parenthesis is added directly to the stack. A right parenthesis empties the stack to the output in reverse order until a matching left parenthesis is found. No parenthesis is sent to the output.

Now comes the tricky part: handling operators. When an operator is encountered, the top element of the stack is examined. The element at the top of the stack will fall into one of three categories: it will be a left parenthesis, it will have a lower precedence than the new operator, or it will have a greater or equal precedence than the new operator.

If the top element of the stack is a left parenthesis, then add the new operator to the top of the stack. If the new operator has a higher precedence than the top element on the stack, add the new operator to the top of the stack. If the top element of the stack is an operator with higher precedence than the new operator, then empty the stack to the output in reverse order until a parenthesis or lower precedence operator is encountered. At that point, add the new operator to the stack.

Figure 3-1 presents a flowchart of the shunting algorithm.

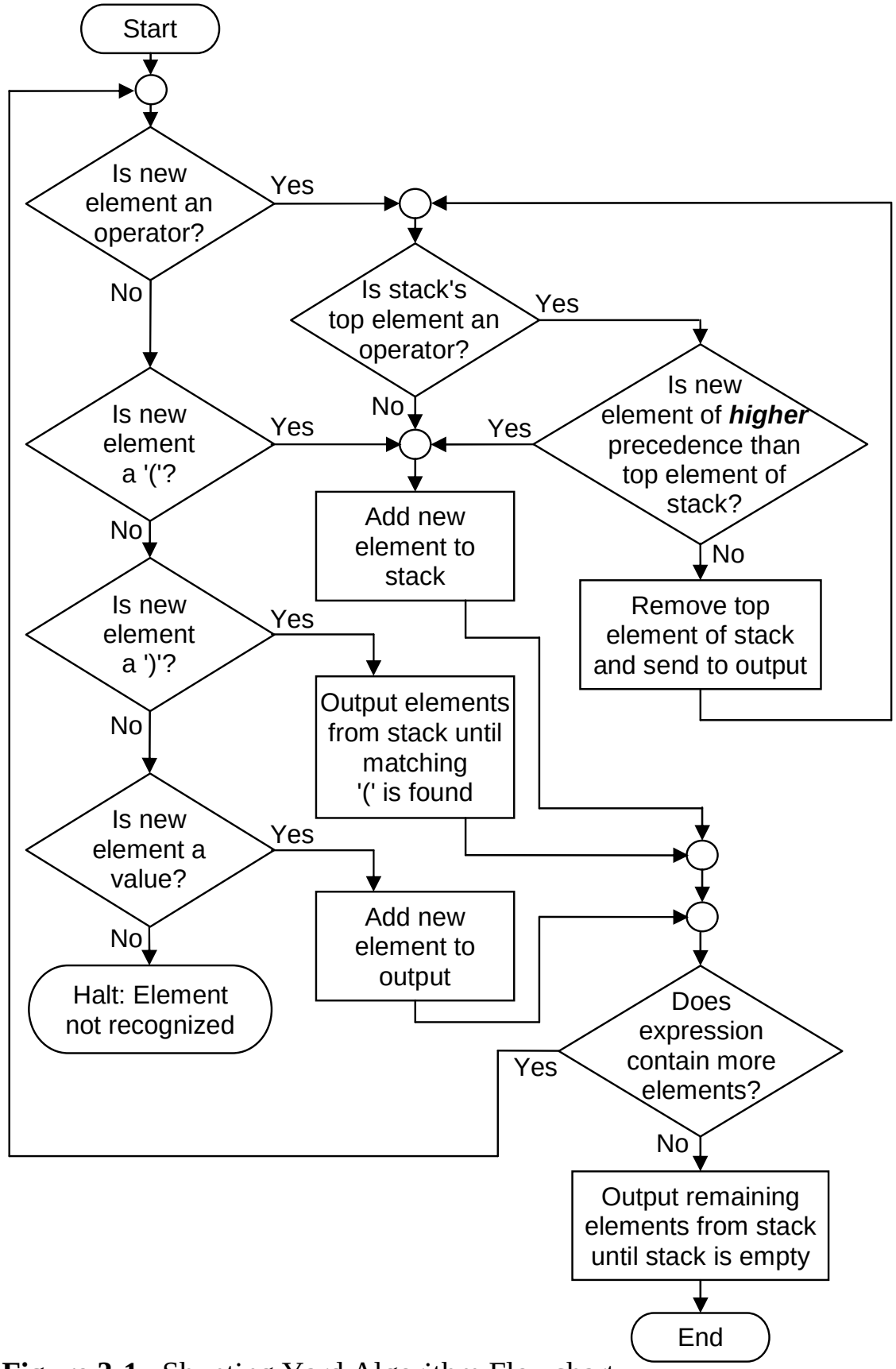

Figure 3-1 Shunting Yard Algorithm Flowchart

Exercise

Convert the expression 3×(4 – 2) – 8÷(7 – 5) + 8 to RPN.

Solution

The table below has four columns. The first column represents the element-by-element sequence of the original mathematical expression. The second column represents the current state of the LIFO stack where operators and parentheses are held until needed at the output. In this column, the element last added appears on the right side. The third column represents the current state of the outputted RPN sequence. The fourth column represents a "running commentary" of how each element is handled.

Element	Stack	Output	Comments
3		3	Values go straight to output
×	×	3	Higher priority operators go to stack
(	×(	3	'(' goes straight to stack
4	×(	34	Values go straight to output
–	×(–	34	'(' is not an operator, – goes to stack
2	×(–	342	Values go straight to output
)	×	342–	The ')' empties the stack up to 1st '('
–	–	342–×	– is lower than ×, × is outputted
8	–	342–×8	Values go straight to output
÷	–÷	342–×8	÷ is higher than –, ÷ goes to stack
(	–÷(	342–×8	'(' goes straight to stack
7	–÷(	342–×87	Values go straight to output
–	–÷(–	342–×87	'(' is not an operator, – goes to stack
5	–÷(–	342–×875	Values go straight to output
)	–÷	342–×875–	The ')' empties the stack up to 1st '('
+	–+	342–×875–÷–	+ is lower than ÷, ÷ is outputted + is not greater than –, – is outputted
8	–+	342–×875–÷–8	Values go straight to output
		342–×875–÷–8+	Empty stack to output

Therefore, the RPN expression for 3×(4 – 2) – 8÷(7 – 5) + 8 is 942–×875–÷8+–.

Let's see if the previous example generates the same result as the mathematical expression. First, compute the result of the expression.

$$\begin{aligned} 3\times(4-2)-8\div(7-5)+8 &= 3\times2-8\div2+8 \\ &= 6-4+8 \\ &= 10 \end{aligned}$$

Next, using the process of successively generating the results from the each *XYf* pattern, compute the result of 342–×875–÷–8+.

342–×875–÷–8+

32× 82÷–8+

64–8+

28+

10

An expression like this, albeit using variables instead of constants, can be evaluated using a 0-address instruction set.

3.3 More on Data

In addition to the number of operands, the size or width of an operand can vary from instruction to instruction. It is assumed that the reader has experience with different data formats such as integers, long integers, floating-point, and so on. Suffice to say at this point that different data types require different numbers of bits. The width of the memory data bus, however, remains constant thereby forcing any operand consisting of more bits than are available from the data bus to be read or written using multiple accesses to the memory data bus.

This means that any operand that is larger than the width of a memory location will have to be partitioned into pieces in order to be stored. These pieces, typically referred to as words, are the width of the memory data bus, and hence, the width of a memory location. This allows the processor to use sequential memory locations to store large values. For example, if a processor with an 8-bit data bus needs to store the 32-bit value $3A2B48CA_{16}$, it would need four memory locations: one each to store $3A_{16}$, $2B_{16}$, 48_{16}, and CA_{16}. When it retrieves the data, it reads all four values and reconstructs the data in one of its registers. The instruction set designer must ensure that the order in which the words are stored remains consistent for both reading and writing, or the value will become corrupted.

Problems occur when the data is transferred between processors that use different orders. Big-endian and little-endian are terms used to identify the observed order in which the partitioned elements are stored. Big-endian means that the first element that is stored in the sequence in memory is the most significant word of the original value. Little-endian means that the first word stored is the least significant word of the original value. The method selected does not affect the starting address, the amount of memory used, or the ordering of items in a data structure.

The reason for discussing this is to show that different operands may require a different number of memory accesses depending on the operand length. Larger operands would require additional memory accesses and increase the amount of time a processor spends fetching data. In addition, if an operation can be performed on different lengths of data, a machine code must be reserved for each supported length for each instruction. For example, if a processor can perform an addition on an integer and on a long integer, two different machine codes must be reserved to explicitly define which addition is being used.

A portion of execution time is also taken when the effective address of an operand must be computed. For example, the base + offset + indirect offset addressing mode requires up to four fetches and two additions simply to compute the address of the data.

In addition to an operand's length, there are also issues surrounding an operand's address in memory. Some architectures restrict an operand's starting address in memory based on the size of the operand. This is referred to as ***data structure alignment***. In general, if an operand contains *k* bytes, then the address at which that operand is stored must be a multiple of *k*. If, for example, an operand is two bytes in length, then the least significant bit of the address must be a 0. If an operand is four bytes in length, then the two least significant bits of the address must be 0.

There are a couple of reasons for this. First, modern processor architectures are set up to read data as blocks or groups of words. If an operand straddles two blocks, then the processor will need to perform two reads to access the operand instead of one. The proper operation of caches (discussed later) also depends on block reads adding another reason to restrict the starting address of a data element.

Second, if data elements are allowed to straddle blocks, the circuitry required to assemble multi-word operands becomes more complex.

More complex circuitry tends to operate slower causing delays or stalls in the execution of an instruction.

Alignment affects more than just the storage of a single data element. Elements within data structures are also required to be aligned on the proper byte boundaries. In addition, it is sometimes necessary to insert meaningless values between multiple operands in order to obey the alignment requirements. This may happen even within a data structure.

3.4 What's Next?

By using the von Neumann architecture as a foundation, this chapter presented the special needs of instructions as they relate to the addressing of their operands. This background is necessary to begin a discussion of the execution of an instruction and the design of an instruction set.

3.5 Problems

1. Describe some of the basic needs of an instruction set's addressing modes.
2. For each of the following addressing modes, identify which components are fixed at assemble time and which are determined at run time.

    a) Immediate addressing
    b) Register addressing
    c) Direct addressing
    d) Register indirect addressing
    e) Indirect addressing
    f) Indexed addressing
    g) Base + indirect offset addressing
    h) Base + offset + indirect offset addressing
    i) Relative addressing
    j) Stack addressing

3. For each of the following situations, select the best addressing mode to be used and justify your answer.

    a) Identifying the start of a string argument in a function call
    b) Adding an array of long integers (each long integer uses 4 memory locations)

c) Maintaining a loop counter
d) Accessing a value within an array of structures

4. The original IAS machine used a ____-address instruction. (Fill in the blank.)
5. The ____-address instruction is typically referred to as an accumulator-based operation. (Fill in the blank.)
6. The ____-address instruction processor is typically referred to as a stack machine. (Fill in the blank.)
7. Convert the following expressions to postfix (RPN) notation.

   a) $5 \times 6 \times 3 \times 2$
   b) $8 - (2 \times 3) - (8 \div 2)$
   c) $(A \div B) + C - (D \times E)$
   d) $A \times B \times C - A \div B$
   e) $((A + B) \times (C + D) \times (E + F)) \div (G + H)$

8. Write the assembly language code to compute each of the above expressions for the 3-, 2-, 1-, and 0-address instruction set processors.