The partial history of computing that is presented here is intended as a starting point for understanding computer architecture. Although electronics and digital circuitry have made great advances, the basic principles of modern computer architecture are similar to those that guided the design of computing devices before 1950.

During the late 19th century and early 20th century, inventors developed computing devices based on the mechanical technologies of their eras. Some featured mechanisms such as pipelining and microcode that are typically associated only with modern computing. Even an innovation like multimedia, which only has recently become viable due to advances in processing power, data compression, and data storage, was anticipated by Lady Ada Lovelace in 1843[1].

Electronics were incorporated into computers in order to improve their speed, reliability, size, assembly, and cost. This fact suggests that implementation has little to do with architecture.

2.1 Blaise Pascal's Calculator

Blaise Pascal's (1623 - 1662) simple arithmetic unit is a good place to begin a discussion of computer architecture. If you are familiar with the operation of a mechanical automotive odometer, then you have a basic understanding of how Pascal's Arithmetic Machine operated.

Unlike modern computers which store values in binary, Pascal's calculator stores a number as a series of decimal digits using a series of wheels each of which can be positioned at one of ten angles corresponding to the digits 0 through 9. Like an automotive odometer, when a wheel passed from the 9 to the 0 position a carry is generated which turns the next highest wheel one tenth of a turn to increment the next decimal place.

The users of Pascal's machine could enter an initial value using dials similar to an old-style rotary phone, then add subsequent values using

[1] "Supposing, for instance, that the fundamental relations of pitched sounds in the science of harmony and of musical composition were susceptible of such expression and adaptations, the engine might compose elaborate and scientific pieces of music of any degree of complexity or extent." (Menabrea & Lovelace, 1843)

the same wheels. The second number would be added to the first and any wheels that turned past 9 to 0 would increment the next highest digit thereby producing the result of an addition. Figure 2-1 presents the basic organization of Pascal's calculator.

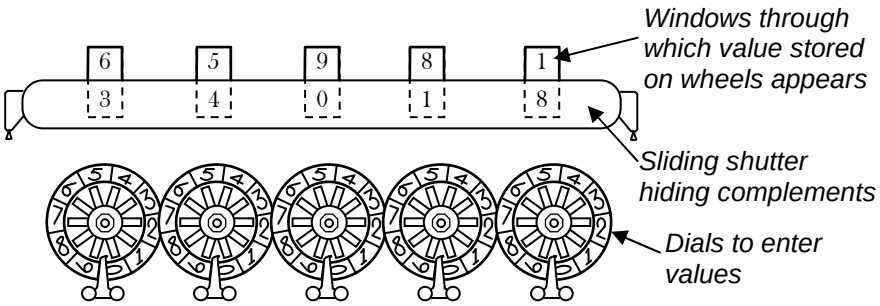


Figure 2-1 Simple Drawing of Pascal's Adding Machine

Machines that can add can also subtract using the *method of complements*. The method is based on the principle that any single digit in any base has a complementary digit that when added to a second digit will generate the difference. Think about how Pascal's calculator represents digits on a dial. Subtracting one digit from another by turning the dial backward can also be accomplished by turning the dial *forward* a complementary number of positions.

Determining a value's complement is a two-step process. First, replace each digit of the number being with its additive complement. In the case of decimal values, that involves subtracting each digit from 9. For example, the complement of 6 is 3, the complement 2 is 7, and the complement of 0 is 9. After each digit of the value has been replaced with its complement, add one to the result.

Let's calculate 7563 – 3867 using addition. To do this, begin by finding the complement of 3867. First, complement each of 3867's digits by replacing 3 with 6, 8 with 1, 6 with 3, and 7 with 2. Then, add one to 6132 to get 6133. Now the result of 7563 + 6133 can be calculated using 7563 + 6133.

```
  6 14 15                      1
  ~~7~~ 5 ~~6~~[1]3          7 5 6 3
– 3 8 6 7                  + 6 1 3 3
  3 6 9 6                  1 3 6 9 6
```

The sum on the right generates a carry out of the leftmost column. This occurs at the leftmost digit of additions where a negative sign needs to be removed from a positive result or when two negative values are added. This carry should be ignored in order to arrive at the correct answer.

Any system that implements this method of subtraction must enter all numbers using a fixed number of digits. The leftmost digit represents a sign digit. In decimal, the sign digit is 0 for nonnegative numbers and 9 for negative numbers. If after a decimal addition the sign digit contains a value other than 0 or 9, the magnitude of the result was too large to be represented with the fixed number of digits.

Pascal's calculator uses 8 dials to represent values. When the leftmost digit is reserved for a sign, the calculator can represent values from 09999999 (positive 9,999,999) to 90000000 (negative 10,000,000). To represent –3867 on this calculator, the complement of 00003867 would need to be taken resulting in 99996133. Adding this to 00007563 yields the desired result:

```
  1 1 1 1 1
  0 0 0 0 7 5 6 3
+ 9 9 9 9 6 1 3 3
  0 0 0 0 3 6 9 6
```

This works for negative results too. To calculate 13293 – 84334, add 13293 to the eight digit complement of 00084334, which equals 99915665 + 1 = 99915666.

```
        1
  0 0 0 1 3 2 9 3
+ 9 9 9 1 5 6 6 6
  9 9 9 2 8 9 5 9
```

Since the leftmost digit is 9, the result is negative. So what is the magnitude of this negative number? It isn't 99928959. That is the complement of the magnitude. Taking the complement of 99928959 using the method described earlier gives 00071040 + 1 = 00071041. Therefore, 13293 – 84334 = –71041.

As an aside, Pascal's calculator displayed its stored value through a set of windows above its dials. A shutter was mounted covering half of each of these windows. Moving the shutter up displayed the decimal

value. Moving the shutter down displayed the digit-by-digit complement. Figure 2-1 shows the shutter's arrangement.

2.2 Charles Babbage's Analytical Engine

With advances in engineering, science, and finance during the 19th century, humans began to depend on large, detailed mathematical tables in order to find the value of complicated mathematical expressions. These tables often contained errors due to mistakes made in the calculations or during typesetting. In an effort to remove human error from these tables, Charles Babbage undertook the design of a series of engines to mechanically calculate and print these tables. His effort is often thought to have resulted in the design of the first automatic computing engines (Swade, 2008).

Babbage's challenge was to combine well-known theories of mathematics with precision machinery to create an engine that would generate mathematical tables without human intervention. One of the outputs of such a machine would be molds for making printing plates.

So what were these theories of mathematics? Let's begin with the difference engine. Imagine that you have two variables, A and B, and a constant, *DIFFERENCE*. Initialize *A* and *B* to 1 and assign a value of 2 to the constant *DIFFERENCE*. Using these elements, create a for-loop that upon each pass through the loop adds *DIFFERENCE* to *B*, then adds B to *A*. Figure 2-2 presents such a loop written in Java.

```
int A = 1, B = 1;
final int DIFFERENCE = 2;
for(int i = 1; i <= 10; i++)
{
      System.out.println(i + "\t" + A);
      B += DIFFERENCE;
      A += B;
}
```

Output

1	1
2	4
3	9
4	16
5	25
6	36
7	49
8	64
9	81
10	100

Figure 2-2 Loop Using Method of Differences to Compute Square

The output from Figure 2-2's loop would be two columns of integers like that shown in the figure's right portion. The column on the left represents the incrementing values of *i* and the column on the right represents the running total *A*. Notice that the values in column *A* are the squares of the respective values of *i*. This shows that a table of squares can be generated using successive additions of a constant *difference*, hence the term "difference machine."

Incorporating another register into the sequence of additions and initializing DIFFERENCE to 6 produces a table of cubes, as shown in Figure 2-3.

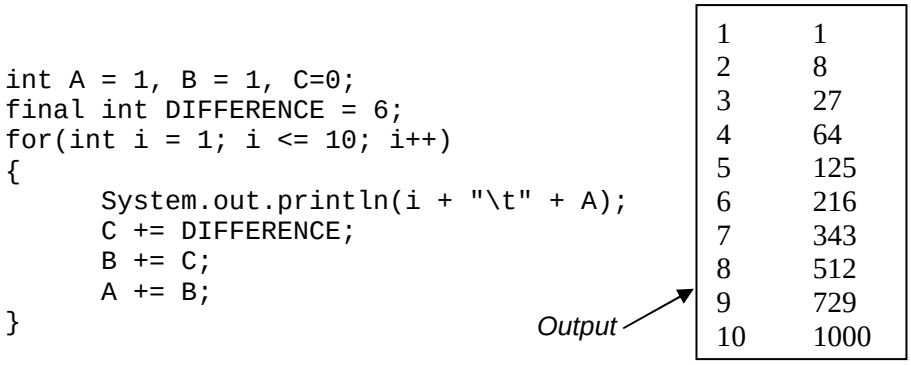


Figure 2-3 Loop Using Method of Differences to Compute Cube

Figures 2-2 and 2-3 illustrate the *method of differences*: a method for computing polynomials using only addition. Since many mathematical functions such as logarithmic and trigonometric functions can be approximated using polynomials, and since a difference engine with n+1 registers can compute polynomials of degree n, then an engine should be capable of generating many types of mathematical tables.

Charles Babbage's 1821 design of his Difference Engine No. 1 was based on this principle. It used columns of wheels, each column of which represented one variable from the for-loop examples and each wheel of which represented a digit of the decimal value, i.e., 1's, 10's, 100's, etc. place.

The ground-breaking achievement of Difference Engine No. 1 was that it was fully automatic. No human thought, understanding of how the machine worked, or understanding of the underlying mathematics was required in order to generate the correct output. Only a physical

cranking of a handle was needed to generate mathematically correct results making it possible to have been produced by a steam engine. (Myhrvold & Swade, 2008)

Difference Engine No. 1, however, was only a start. Its capabilities were far beyond the calculating machines of the day, but its application was limited to the solution of polynomial expressions. The design that followed, Babbage's 1838 Analytical Engine, came close to the architecture of a modern, general-purpose computer. This engine included the following features (Myhrvold & Swade, 2008):

- ***An internal von Neumann architecture:*** control and instructions (the mill) were separated from the data (the store)
- ***Memory:*** at least 100 data memory locations (the store)
- ***Buffered parallel bus:*** transfer of data between mill and store
- ***General-purpose programming:*** serialized punch card input for general-purpose programming
- ***Pipelining:*** a serial fetch/execute cycle where the fetch unit and execute unit could operate simultaneously
- ***Conditional branching:*** primitive if-then statements using conditions such as "is like"
- ***Looping:*** iterative looping accomplished through the physical connection of the programmable cards in a ring
- ***Microprogramming:*** reconfigurable barrels inside the mill that converted punch card instructions into sequences of microcode.
- ***Carry look-ahead addition:*** carries generated across multiple decimal digits in order to reduce the execution time of an addition
- ***High precision:*** 50 to 100 decimal digits
- ***Parallel processing:*** the use of independent arithmetic units
- ***Advanced I/O:*** output was available in numerous forms including printed, stereotype printing plate, punch card, and graphical

Babbage's Analytical Engine called for a phenomenal effort in machining and manufacture. A single 40-digit decimal value of the Analytical Engine would be stored on an axis of three-inch tall rotating cylinders (one for each digit) measuring a total of ten feet in height. It is estimated that the *store* (memory) unit alone would contain 100 axes for storing numbers, two axes for buffering numbers transferred to and from the mill (I and "A), and an axis for a counter. This would require the store to measure from 10 to 20 feet in length (Bromley, 1998).

The *mill* (control unit) contains two accumulator axes (A and 'A), three axes to handle carry look-ahead for addition and subtraction or the upper half of a multiplication result (F, 'F, and "F), nine table axes used for multiplication and division (T1 through T9), and four additional axes used for temporary registers and data transfer. Add to this the mechanics for the microprogramming control unit, error handling, and input and output and you have a machine that is the size and weight of a small locomotive (Bromley, 1998).

The implementation of the Babbage Analytical Engine, however, is not important to this discussion. What is important is the programmer's view of the engine. First, the Analytical Engine separated data (the store) from control (the mill). In addition, the Analytical Engine separated data from the instruction sequence, an architecture later to be known as the Harvard architecture (Bromley, 1998). This allowed data to be a different format and width than the instructions. It also allowed data and instructions to be fetched in parallel thereby improving performance. The problem was that additional connections were required for the two independent connections to the processing unit.

As discussed earlier, data was stored on rotating wheels lined along an axis. Since a rotating shaft can have hundreds of positions as easily as it can have two, the binary numbering system is of little importance to a mechanical system such as Babbage's. This aside, Charles Babbage did experiment with various bases including 2, 3, 5, 12, 16, and 100 in an effort to determine which base resulted in the fastest and least amount of hardware. In the end, he settled on decimal to eliminate the need for converting between stored values and input from and output to the user (Bromley, 1998).

In order to make his machine programmable, Babbage borrowed two technologies from his own times. First, each instruction was encoded on its own punch card, a technique borrowed from the Jacquard loom. A set of cards was strung together using ribbons allowing the cards to act much like a paper tape. Two sets of cards were used, one for operations and one for variable values.

The other technology borrowed by Babbage was used to interpret the instructions found on the cards. Each instruction corresponded to a sequence of operations in the mill. These operations were controlled by a set of studs screwed onto a barrel much like that used in a music box. The sequence of operations that defined an instruction could be reconfigured by changing the position of the studs. This process of

implementing an instruction through a sequence of micro-operations is in use today and is known as microprogramming (Bromley, 1998).

Babbage's instruction set itself is a bit more difficult to describe as it seems that nowhere is one clearly defined. However, through the machine's microprogramming ability and its ability to execute a limitless number of instructions/cards, Charles Babbage had designed a universal calculating machine with capabilities and features found in many modern computers (Bromley, 1998).

2.3 Konrad Zuse's Z1

From 1936 to 1938, Konrad Zuse constructed his Z1, an automaton he had designed in order to perform the time-consuming calculations necessary in his occupation as a civil engineer. (Zuse P. , 2001) It incorporated a number of important features still used in modern computing including a floating-point binary numbering system, an addressable memory to store data, distinct decimal input and output units, an arithmetic unit with a carry look-ahead adder, and a control unit that used a two-stage execution pipeline. The machine itself was mechanical although Zuse later developed machines with electromechanical relays. (Rojas, 1997) His third machine, the Z3, was identical in architecture to the Z1 except that it replaced the unreliable mechanical elements with electromechanical relays and added a square root function (Rojas, 1997).

Like Babbage's Analytical Engine, the Z1 used a Harvard-style architecture executed programs directly from a punch tape reader (35mm standard movie film) and stored data in a separate memory. In the case of the Z1, storing the programs in memory was avoided because it would have been too expensive to increase the size of memory to contain the programs. (Zuse P. , 2001)

The Z1's memory was connected to the arithmetic unit through a parallel bus that passed 22-bit[2] data between the 64 memory locations and the arithmetic unit's two floating-point registers, R1 and R2. These registers functioned as an accumulator (R1) and a temporary register (R2) (Zuse P. , 2001).

Zuse's floating-point data format is remarkably similar to the IEEE Standard for Floating-Point Arithmetic (IEEE 754) in common use

[2] Zuse did not refer to the individual binary digits as "bits". Rather, he described them as "yes/no status". (Zuse K.)

today. The most significant bit is a sign bit for the value. The next seven bits contain the base-two exponent coded in two's complement representation. The last fourteen bits are the significant without the "hidden bit", i.e., the most significant bit of the significant, which is always a one for non-zero values. Zero was represented with an exponent of –64, a binary 1000000, with any value for the significant. (Rojas, 1997) Figure 2-4 presents the binary floating-point format of the Z1 data.

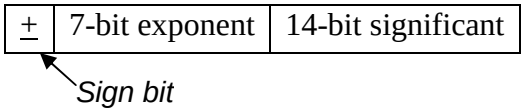


Figure 2-4 Zuse's Floating-Point Representation

Figure 2-5 shows $3{,}245_{10} = 110010101101_2 = 1.10010101101 \times 2^{11}$ in Zuse's Z1 representation.

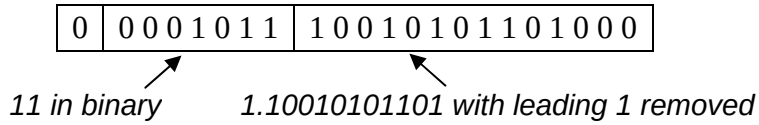


Figure 2-5 Zuse's Floating-Point Representation of $3{,}245_{10}$

The Z1 instruction set consisted of eight 8-bit instructions including two for memory transfers and one each for input and output. The remaining instructions represented mathematical operations between the two registers. Addition, subtraction, multiplication, and division combined R1 and R2, placed the result in R1, and cleared R2. Zuse later added a ninth instruction for the Z3 that took the square root of R1 and returned the result to R1 (Rojas, 1997).

The operand field of the memory load instruction held the address from which data was to be loaded, but did not identify the destination register. This is because the first load would always load R1 with subsequent loads loading R2. The operand field of the memory store instruction also did not identify a register. All memory stores would store R1 to memory and then clear R1. The next load would then be considered "first" and load R1 again (Rojas, 1997).

User input was accomplished using a four column by ten row keyboard that allowed the user to enter a decimal digit for each of the four digits of the significant. A second keyboard allowed the user to enter a decimal exponent from –8 to 8. The "read keyboard" command read the keyboard's settings, converted them to binary, then loaded the value to R1 and cleared R2 (Rojas, 1997).

A "display result" command converted the binary value contained in R1 to decimal, display the decimal value on a set of lamps, and then clear R1 and R2. The output lamps were arranged much like the input keyboard, i.e., digits for the significant and an indicator for the exponent. The significant output allowed for 5 digits instead of 4, the most significant of which could only be a 0 or a 1. "Read keyboard" and "display result" instruction also halted the Z1 to give the operator a chance to enter the next number or record a result.

Table 2-1 presents a summary of the Z1 instruction set along with each instruction's opcode and the number of cycles required to execute the instruction (Rojas, 1997).

Table 2-1 Z1 Instruction Set (Rojas, 1997)

Instruction/Description	Opcode	Cycles
Lu – read keyboard	01 110000	9 to 41*
Ld – display result	01 111000	9 to 41*
Pr z – load from address z	11 $z_6z_5z_4z_3z_2z_1$	1
Ps z – store to address z	10 $z_6z_5z_4z_3z_2z_1$	0 or 1
Lm – multiplication	01 001000	16
Li – division	01 010000	18
Ls1 – addition	01 100000	3
Ls2 – subtraction	01 101000	4 or 5

* - depending on exponent

What the Z1 lacked was program flow control. There were no conditional branches. Loops, therefore, could only be implemented by connecting the paper tape's ends to form a physical loop in the code. There was also no way to program decision structures such as if or if/else statements (Rojas, 1997).

With the completion in 1941 of his Z3, Zuse created a reliable, programmable computer with a binary floating-point number system. The next step was conditional branching and storable programs.

2.4 IBM Automatic Sequence Controlled Calculator

Although numerical methods such as the method of differences can reduce complex mathematical calculations to the operations of addition, subtraction, multiplication, and division, the volume of operations and the requirement of storing past results demands the use of calculating machinery. It is for this reason that the Automatic Sequence Controlled Calculator (ASCC) was developed, the result of a collaboration between Harvard University and the International Business Machines Corporation (IBM) (Harvard University Computation Laboratory, 1946). After a five-year design and development process, IBM shipped the ASCC to Harvard University in February of 1944 and in August of 1944 completed final assembly and presented it to the University (IBM).

Also known as the Harvard Mark I, the ASCC separated data storage from processing just like Babbage's Analytical Engine. The ASCC's processing unit consisted of a central multiply and divide unit, an array of seventy-two accumulators (referred to as counters in the Harvard documentation) for adding and temporary storage, and an automatic sequencer that controlled the machine as a whole. In addition, tables representing $\log_{10}(x)$, 10^x, and $\sin(x)$ were included in order to improve performance by removing the need to compute these common functions. Up to eleventh order interpolation could also be performed by using one of the ASCC's three interpolation units. (Harvard University Computation Laboratory, 1946).

The ASCC was an electromechanical calculator measuring 51 feet long by eight feet high and weighing about five tons. The machine's mechanical components required a four horsepower electric motor to drive them. Data values were entered using one of two card readers, and constants could be entered using a set of rotary knobs. Instructions were entered using a separate mechanism called the automatic sequencer. This device used a drum over which ran a control tape with rows of punched holes representing instructions. The ASCC output consisted of an automatic card punch and two automatic typewriters.

The ASCC's operation was controlled by a perforated paper tape that was fed into the sequence control unit. The tape reader was

equipped with a row of 24 sensing pins that read the perforations in a single line. Since each position either had a hole or no hole, there was an upper limit of 2^{24} (16 Meg) possible binary values that could be input to the sequencer on a single line. Due to limits on the set of allowable combinations of operations and data sources, the number of valid 24-bit inputs was considerably smaller than 2^{24}.

Each row of 24 holes was partitioned into three 8-bit groups, A, B, and C. Group A identified the source register, referred to in the Harvard documentation as the "out relays". Group B identified the destination register, referred to in the Harvard documentation as the "in relays". The last group of eight bits, the C group, defined an operation to be performed to generate the final result of the operation.

Groups A and B referred to elements from one of two sets of registers: a first set of storage registers whose values could be varied, and a second set of constant registers.

The ASCC had 72 storage registers numbered 1 to 72. Each of these registers consisted of 24 ten-position electro-mechanical counter wheels representing a 23-digit decimal number with an initial sign digit. In keeping with the method of complements representation, the sign digit was set to 0 for a non-negative value, 9 for a negative value, and any other digit for an overflow. Figure 2-6 shows a partial set of counter wheels for register 16.

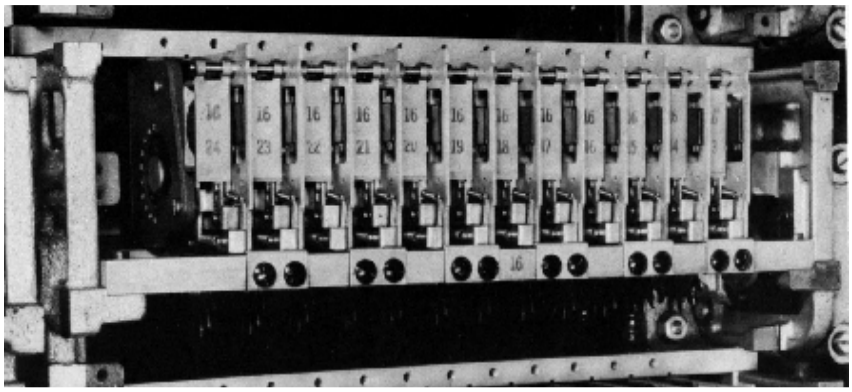


Figure 2-6 IBM ASCC Storage Counter Wheels (Harvard University Computation Laboratory, 1946, p. 16b)

The ASCC represented each storage register using a code based on its register number. These codes, which were used to reference the registers in the ASCC's programs, consisted of a list of those positions in the number's binary representation that correspond to 1's, with the least significant binary digit being referred to as position 1. For example, register 37_{10} (00100101_2) would have holes in positions 6, 3, and 1 thereby giving it a code number of 631. Register 71_{10} (01000111_2) would be identified with 7321.

The wheels of these accumulator registers operated much like the wheels of Pascal's calculator. A number or its complement could be added to a register and the register's hardware would handle the addition along with any generated carries (Harvard University Computation Laboratory, 1946)

The ASCC's constant registers consisted of sixty 24-digit decimal vales to be used in the computations. Each digit of these registers was set manually using a 10-position dial, with negative values represented by their complements. Figure 2-7 presents an image of the most significant 6 digits of constant registers 14 through 18.

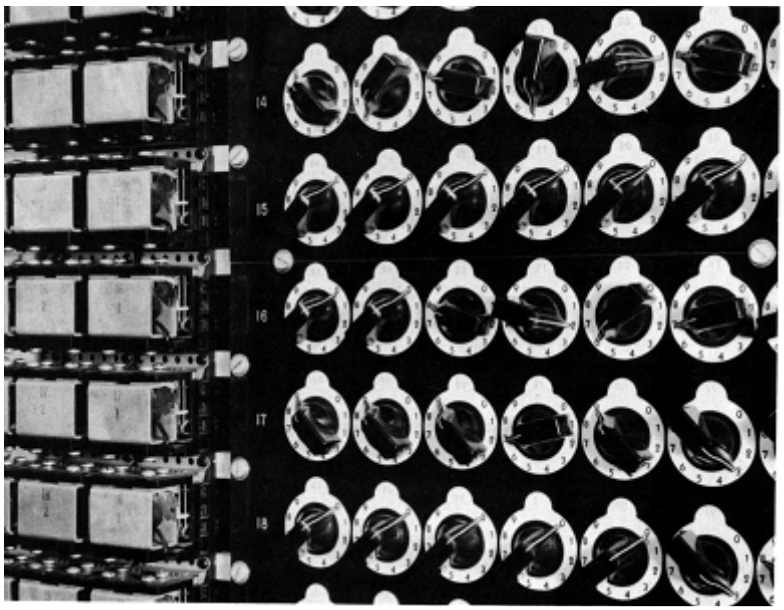


Figure 2-7 IBM ASCC Constant Input Dials (Harvard University Computation Laboratory, 1946, p. 16a)

The constant registers, which were numbered 73 through 132, used the same encoding scheme as the storage registers. This means that the registers are referred to with the codes 741 ($73_{10} = 01001001_2$) through 83 ($132_{10} = 10000100_2$).

The typical ASCC command took the value contained in the register identified by the A group (the source), added it to the register identified by the B group (the destination), then stored the result in the B group register. For example, Figure 2-8 shows a pattern of holes in a row on the tape that would instruct the ASCC to add register 23_{10} (binary 00010111/code 5321) to register 40_{10} (binary 00101000/code 64). Note that the C group has a hole punched at position 7. This directs the ASCC to continue to the next instruction without halting.

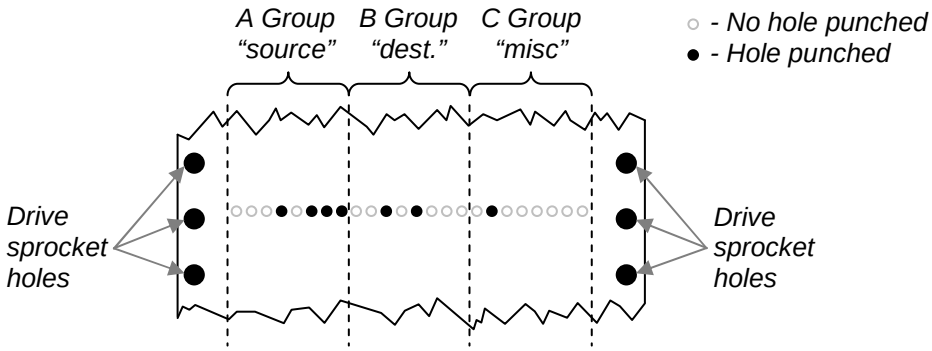


Figure 2-8 ASCC Addition Instruction Example

The settings of the eight bits or holes in the C group allowed for special operations to be done either during an addition or instead of it. For the most part, each C group bit denoted a specific operation, while combinations of these bits defined more complex instructions. Table 2-2 presents some C group codes and their corresponding operations.

To perform a subtraction, we need to complement the source before adding it. This would involve setting the lower C group bits to 32. Figure 2-9 shows the instruction to subtract register 23 from register 40 with the result being placed back in register 40.

Since clearing a register involves adding its value to its complement, the register to clear must be identified both in the A and B groups. While a clear could be specified using the C group code 732, the command's prevalence led the ASCC's designers to denote a clear using a blank or a hole only at position 7 (the continue command). The

designers felt that it would ease coding requirements to not require the programmer to use the 32 code for the complement. This is only true if both the A and B groups refer to the same register.

Table 2-2 Subset of ASCC Instruction Set (Harvard University Computation Laboratory, 1946)

Code/holes	Instruction/Operation
32	Take complement of source before adding to destination
7 with other operations	After execution, go to next instruction. If command has no 7, the machine is halted after the instruction.
7 with no other operations	If A and B are the same, clears a register by adding complement of register to itself. Addition otherwise.
2	Take absolute value of source before adding to destination register
1	Take negative absolute value of source before adding to destination register
51	Output value to punch card. Halt machine if no card present.
6	Print output to automatic typewriter, (752 or 7521 in A group)
87	Halt machine at next line of code.
741	Read into EIO counter

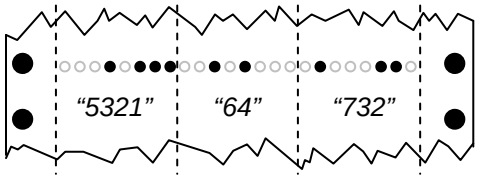
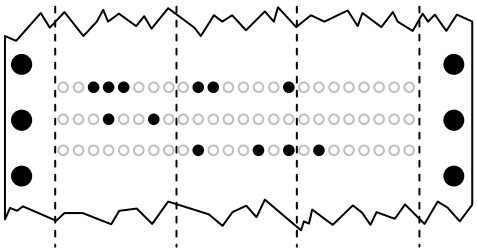


Figure 2-9 ASCC Subtraction Instruction Example

The ASCC also performed multiplication and division. Multiplications used two 24-digit registers as sources and a 46-digit register pair for the result. Since multiplication used four registers, the command itself required multiple rows on the tape. The first row used

the A group to identify the first source register and the B group code "761" to signal the multiplication was coming. The C group was left blank. The second row used only the A group which identified the second source register. The third row used the B group to identify one of two register pairs that were capable of handling 46 decimal digit results. The continuation command 7 was only required in the third row as the 761 in the first row's B group signaled that the operation would span three rows.

Figure 2-10 presents a sample section of code that multiplies the value in register 56 by the value in register 18, stores the result in register 69, then continues to the next instruction. The quotation marks around the "761" and "7" are added to indicate they are codes and not decimal values.

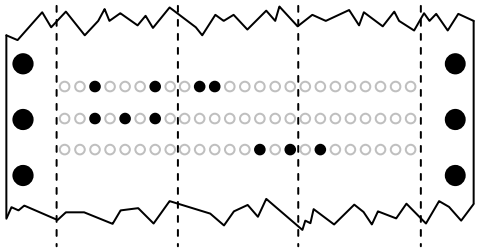


Group A	Group B	Group C
56	"761"	
18		
	69	"7"

Figure 2-10 ASCC Multiplication Instruction Example

Like multiplication, division used three rows/instructions. The difference is that the code used in B group for the first row/instruction is "76" instead of "761". The operation places the remainder in the register that contained the original dividend. This is because a division is usually performed with successive subtracts of the dividend until the result is less than the divisor. Figure 2-11 presents a sample code that divides the value in register 34 into the value in register 42. Once the division is complete, the quotient is moved to register 5 while the remainder is left in register 42.

Similar instruction sequences allowed the programmer to compute logarithms (code 762), exponentials (code 7621), interpolations (code 763), and sine functions (code 7631). Although the larger instruction set of the ASCC sets it apart from Zuse's Z1, the Z1 had an advantage over the ASCC in that the ASCC did all of its computation using a fixed-point system rather than the floating point system.



Group A	Group B	Group C
34	"76"	
42		
	5	"7"

Figure 2-11 ASCC Division Instruction Example

Some of the ASCC's registers were assigned special purposes. The upper and lower halves of register 71 (code 7321), the "multiple-in-out" counter, could be treated as two independent 12-digit values. When used with other registers, it could double the ASCC's storage capacity at the cost of lower accuracy.

Additional carry circuitry between registers 68 and 69 and between registers 64 and 65 created two pairs of "ganged counters" allowing for 46 decimal digit results. Registers 68 and 64 were considered the most significant halves of the pairs. The sign bit was duplicated in the most significant two digits of the final result.

To differentiate between their use as a 24 digit register/counter or as a ganged register/counter pair in code, an '8' was appended to their code, i.e., the most significant digit was set to specify ganged mode. For example, to refer to register 69 as a single, 24-digit register, the code 731 was used. When referring to register 69 as part of the ganged pair with register 68, the code 8731 was used. The operations were still in 24-digit mode; i.e., an addition would take two instructions, the first adding the lower halves and the second adding the upper halves along with the carry from the previous addition.

Registers 70 and 72 could be used to support conditional programming. Register 70 (code 732), the "choice counter", allowed for a primitive conditional operation. When the C group was set to code 432, the ASCC added or subtracted the registers referred to using the A and B groups, according to whether the choice counter contained a positive or negative value.

Register 72 supported more advanced conditional programming structures. Basic conditional programming structures consist of a comparison between two values (usually a subtraction) and a decision

as to which path through the code to take based on the comparison's result. The ASCC used a special instruction, code 64 in the C group, to evaluate the subtraction result in register 72.

Unlike modern conditional branches, the ASCC merely continued or halted operation based on a comparison's outcome. The process was to load register 72 with a value, add the negative absolute value from a second register to register 72, then execute operation 64. If the result of the subtraction was equal to or less than zero, the ASCC would halt allowing a new tape to be loaded. Otherwise, operation continued.

With its separate instructions (tape) and data (punched cards or constant registers), the ASCC had the same architecture as Babbage's Analytical Engine and Zuse's Z1. The ASCC, however, gets the honor of having the architecture named after it. Computers that separate data and instruction storage are commonly referred to as having a *Harvard architecture*. Until memory became cheaper and more compact, stored program architectures were not practical.

2.5 Early Stored Program Machines

Separating instructions from data memory has benefits such as allowing independent formats and simultaneous access of instructions and data. For early machines, however, the separation of data and code may have been more a necessity than an attempt to improve performance. Memory was too expensive and required too much physical space to waste on instructions. Instructions were relegated to the more cost effective and reliable format of paper tape (Zuse P. , 2001).

Paper tape, however, had drawbacks, the worst of which was that it was incapable of implementing any decision structure other than to halt the machine so that the tape could be repositioned or a new tape could be loaded. The solution to this was to create a machine that had enough addressable memory to store both instructions and data.

2.5.1 Manchester Small-Scale Experimental Machine

In 1948, a proof of concept for an electronic memory called the Williams Tube resulted in the Manchester Small-Scale Experimental Machine (SSEM), the first stored-program computer. The SSEM had a 32×32-bit memory, a 32-bit accumulator for intermediate results, and a register called "the control" to contain the address of the current

instruction. During execution, the control was concurrently loaded with the instruction itself (Burton, 1997).

Both instructions and data were stored using 32-bit words. The instruction contained a 13-bit address (bits 0 to 12) and a 3-bit function field (bits 13 to 15). Because the SSEM had a 32-word memory, five bit positions (bits 0 to 4) were used for the address.

Figure 2-12 presents the layout of an SSEM instruction. The leftmost bit, which was referred to as bit 0, represented the least significant digit. It was common at the time of the SSEM development to view bits as if they were arranged along an X-axis with magnitude increasing from left to right. This was also true for numeric values, which were represented in 32-bit two's complement form (Burton, 1997).

Bit position:	0	1	2	3	4	5 .. 12	13	14	15	16 .. 31
Function:	Memory address					Unused	Function			Unused

Figure 2-12 SSEM Instruction Layout

The SSEM had a small instruction set, each function of which was identified using three bits. One function accessed the arithmetic unit, which was capable only of subtraction and negation. For flow control there were two unconditional jump functions, a conditional jump function, and a stop function. One function for loading and one function for storing allowed the SSEM to access data in memory. Table 2-3 presents the SSEM instruction set along with the binary function codes for programming it (Burton, 1997).

Program flow was controlled with the control register, CI. Before an instruction was loaded, CI was automatically incremented by one to the next memory address. This had two effects. First, if the SSEM initialized CI to 0 before running a program, the first instruction to be executed would actually be at location 1. Second, the destination address for jump instructions was given as one less than the desired address.

The CMP function (011_2) also affected CI. If a CMP function was encountered and the accumulator contained a negative value, CI was incremented an additional time thereby skipping the subsequent instruction (Burton, 1997).

Table 2-3 SSEM Instruction Set (Burton, 1997)

Binary function	Modern Mnemonic	Instruction/Description
000	JMP	Load CI with address stored in location pointed to by instruction (indirect jump)
100	JRP	Retrieve offset stored in location pointed to by instruction and add it to CI (relative jump)
010	LDN	Copy to accumulator the negated contents of the memory location pointed to by instruction
110	STO	Copy contents of accumulator to memory location pointed to by instruction
001	SUB	Subtract from accumulator the contents of the memory location pointed to by the instruction
101	-	Not defined (If used, SSEM performed SUB function)
011	CMP	Skip the next instruction if the content of the accumulator is negative
111	STOP	Halt the machine

The SSEM's two jump commands are considered special cases in modern computing. The JMP function (000_2) is now known as an indirect jump. This means that the memory address referenced by the instruction contains the new address to be jumped to unconditionally. The JRP function (100_2) is now known as an indirect relative jump. This means that the value stored at the address pointed to by the instruction represents an offset from the current instruction.

Although JRP is not necessary for a machine of SSEM's simplicity, it was added because its cost was trivial and it allowed for relocatable code. For example, a loop with an instruction that says, "jump 5 memory locations back" instead of using a hard coded address can be placed anywhere in memory (Burton, 1997).

Programming the SSEM could be quite difficult due to its limited number of instructions, 32-word memory, and lack of some common instructions such as addition or a direct load from memory. For example, to load the accumulator with a value from memory took three steps: function LDN (010_2) to load the accumulator with the negated value in memory, function STO (110_2) to store that negated value to a temporary memory location, then a second LDN to retrieve the value

from the temporary memory location and negate it, thereby arriving at the original value. Similarly, to add two numbers the SSEM had to load the negative of the first value (A = –x), subtract the second value (A = –x – y), store the value in a temporary memory location, then load the negative of the newly stored value back into the accumulator (A = –(–x – y) = x+y).

Figure 2-13 presents a simple program to compute the quotient and remainder from a dividend and a divisor. It requires that the dividend (5,267 in the sample code) and divisor (43 in the sample code) be positive and that the dividend be larger than the divisor. The resulting quotient, 122, is stored in memory at line 27 while the remainder, 21, is placed at line 24 where the starting dividend was stored. Once again, this is because the process of division involves repeated subtractions of the dividend until the dividend is less than the divisor and hence becomes the remainder.

The success of the SSEM along with its stored-program architecture led to the development and implementation of the Manchester Mark 1. The Mark I, which had a 40-bit data word size and a 20-bit instruction, took the accomplishments of the SSEM to a much greater level.

2.5.2 Princeton IAS Machine

After having worked on the Electronic Numerical Integrator and Calculator (ENIAC) with John W. Mauchly, a professor at the University of Pennsylvania's Moore School of Electrical Engineering, and J. Presper Eckert, Mauchly's graduate student, John von Neumann, a mathematician at Princeton's Institute for Advanced Study (IAS), developed a new design for a stored program architecture. This work was motivated by the difficulties with programming the ENIAC. The ENIAC was programmed manually by setting switches and routing cables, an arrangement that was time consuming and did not allow for the storing of programs for later reload and execution. With Mauchly and Eckert's input, von Neumann wrote a report titled, "First Draft of a Report on the EDVAC," which proposed a machine that stored instructions in memory alongside the data (von Neumann, 1945).

In the report, von Neumann identifies the requirements of the ENIAC's successor, the Electronic Discrete Variable Automatic Computer (EDVAC). This report was the first to describe a computing machine in the abstract sense rather than as a description of the electronic circuitry (Smith, 1989).

	Instruction	Binary code	Comment
0	0000000000000000…0		CI is incremented over this to 1
1	LDN 24	00011...010	Load negated dividend
2	STO 25	10011...110	Temporarily store negated dividend
3	LDN 25	10011...010	Load positive dividend
4	SUB 26	01011...001	Subtract divisor from dividend
5	CMP	00000...011	If result < 0, dividend was remainder
6	JRP 28	00111...100	Jump over stop using const. 1 at 28
7	STOP	00000...111	Program completed
8	STO 24	00011...110	Store updated dividend
9	LDN 27	11011...010	Load negated running quotient
10	SUB 28	00111...001	Subtract 1 to "increment" neg. quotient
11	STO 25	10011...110	Temporarily store negative quotient
12	LDN 25	10011...010	Load positive quotient
13	STO 27	11011...110	Store positive quotient to proper line
14	JMP 29	10111...000	Jump to line 1 by loading CI with 0
…			
24	1100100100101000…0		Positive dividend initialized to 5,267
25	0000000000000000…0		Temporary memory location
26	1101010000000000…0		Positive divisor initialized to 43
27	0000000000000000…0		Running quotient initialized to 0
28	1000000000000000…0		Constant 1 used in lines 6 & 10
29	1000000000000000…0		Constant 0 used in line 14
…			

Figure 2-13 A Sample SSEM Program to Perform Division

First, because of the machine's nature, i.e., its use as a computing device, it must contain specialized components capable of addition, subtraction, multiplication, and division. This element of the machine, identified as CA in the report, might also include functions such as the square root, cubed root, sgn, absolute value, natural logarithm, base 10 and 2 logarithms, and sine along with some of their inverses (von Neumann, 1945).

The machine should also have a logical control component (CC) that can step through a sequence of instructions to execute and activate the appropriate sub-systems according to the instructions. von Neumann wrote that the order of the instructions and not the instructions themselves should define a particular application's solution. This is critical in order to keep the device flexible and capable of general

purpose application. This also implies that the instructions should be kept separate from the control unit, i.e., stored in a separate memory (von Neumann, 1945).

In order to handle problems beyond a trivial nature, the machine must be able to store a significant number of instructions in its memory. In addition to instructions, the memory might also be used to store intermediate results for complex operations, parameters for expressions, tables of constants, boundary conditions, conditions for interpolation, and arrays of values for sorting or statistical evaluation. These requirements imply the machine should have considerable memory (M).

von Neumann notes that it would be beneficial to create a single memory that could be reallocated to differing needs. He does, however, make an exception. Because of their short duration, it might be convenient to move the storage of intermediate results to the CA component (von Neumann, 1945).

In order to interact with humans, the device must have a medium (R) that can be acted upon and/or sensed by a user. If this component were to store results on material such as magnetized metal tape or wire that could later be re-entered, R would also have the properties of static memory and hence could be the natural solution for long term storage.

An input interface (I) would be needed to receive information from R and transfer it to the machine. von Neumann suggests that it would be best to transfer user input directly to memory and never into the control or arithmetic unit. Similarly, the machine should have an output interface (O) that delivers computed results from the machine back to the user medium R. Once again, von Neumann suggests that it would be best to make the transfers directly from memory and not from CC or CA (von Neumann, 1945).

Although the von Neumann documentation does not provide a block diagram for the organization of the proposed EDVAC machine, Figure 2-14 presents an interpretation of his work.

John von Neumann then collaborated with Arthur Burks and Herman Goldstine at IAS to develop a new machine based on the EDVAC design, a machine later known as the IAS machine. The IAS machine, described in their report titled, “Preliminary Discussion of the Logical Design of an Electronic Computing Instrument,” had the same architecture as that described in the EDVAC report although the later paper went into more detail regarding implementation including a

description of the instruction set (Burks, Goldstine, & von Neumann, June 1946).

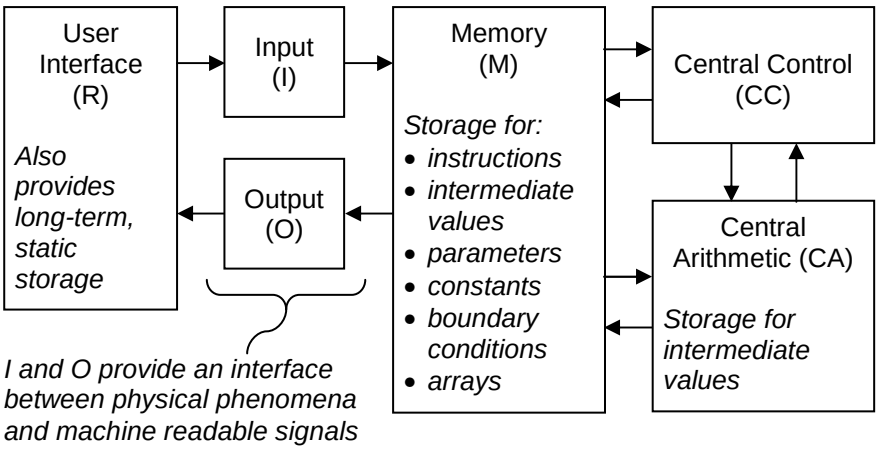


Figure 2-14 Block Diagram of the von Neumann Architecture

According to this paper, the EDVAC machine was to be designed according to the following guidelines.

- The machine's instructions were to be contained in memory.
- A method was to exist so that the machine could distinguish an instruction from a number thereby allowing both to be stored in the same memory.
- Although values could be located anywhere in memory, instructions were to be ordered linearly, i.e., an instruction in place n in memory was to be followed by the next instruction in place (n + 1).
- The instruction set was to include instructions that could exercise each of the arithmetic unit's operations, transfer data between the machine's elements and transfer data to the output units and from the input units.
- Given the need for repetitive tasks, the instruction set was to contain a conditional transfer order capable of deciding which of two sequences of instructions to execute. One suggested method on which to base the decision was the sign of a number.
- Since a conditional transfer instruction allowed for one of two possible sequences of instructions to be executed and since those

sequences might need to come back together at a common point, the instruction set should also have contained an unconditional transfer order.

- Because it had been determined that the IAS machine's implementations of multiplication and division required the ability to shift the values in the registers right and left respectively, it was allowed that these operations should also be included in the instruction set.

As implemented, the Princeton IAS machine had a 1,024 word binary memory, each location of which could store a 40-bit word. When used to store data, the 40-bit location used two's complement representation of a fraction, i.e., a fixed point value between -1 and 1. When used to store an instruction, each 40-bit location stored two 20-bit instructions identified as "left" and "right". The left instruction was executed first (Goldstine, Pomerene, & Smith, 1954).

Each 20-bit instruction was partitioned into four parts: a 10-bit memory address, a step digit used to halt or continue operation, an 8-bit operation code, and a spare bit that identified special operations. The 10-bit address allowed the IAS machine to access any location in its 1,024 word memory. The step digit, if set to 0, halted the machine after the execution of an instruction without incrementing the program counter. Otherwise, if the step digit was set to 1, the program counter was incremented and the next instruction was executed. The 8-bit opcode defined the operation to execute. An instruction’s last bit was 0 except in the case of a special order instruction (Goldstine, Pomerene, & Smith, 1954). Figure 2-15 presents the layout of the IAS instruction word

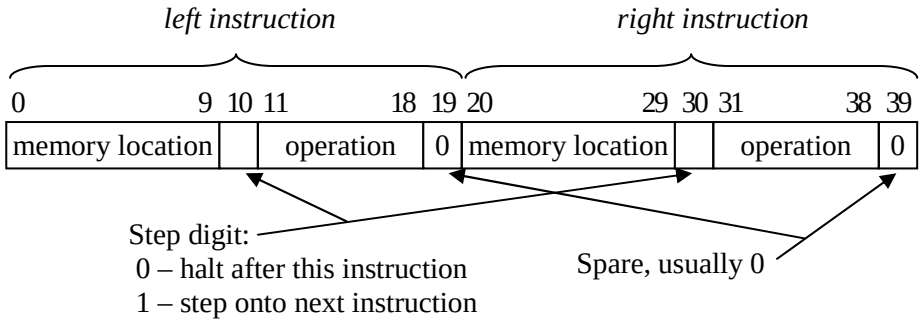


Figure 2-15 Princeton IAS Machine Instruction Layout

The IAS machine had seven registers for storing binary values. Two 40-bit registers in the arithmetic unit, CA, maintained the sources and results of arithmetic operations.

- ***Accumulator (AC)*** – This 40-bit register (also referred to as RI in IAS documentation) held the result of an arithmetic operation, the upper half of a multiplication result, or the remainder of a division. It also held the multiplicand for a multiplication or the dividend of a division.
- ***Multiplier Quotient (MQ)*** – This 40-bit register (also referred to as RII in IAS documentation) held the lower half of a multiplication result or the quotient of a division. It also held the multiplier for a multiplication.

Five registers in the control unit, CC, maintained the step-by-step sequencing of instructions from memory.

- ***Program Counter (PC)*** – This 10-bit register contained the address of the next 40-bit instruction pair to fetch from memory.
- ***Memory Buffer Register (MBR)*** – This 40-bit register contained the word (datum or instruction) to be written to or most recently read from memory.
- ***Memory Address Register (MAR)*** – This 10-bit register contained the address for a transaction to or from memory. If the next read was to be an instruction pair, MAR contained the value from the PC register. If the next transaction was to be a data value, MAR contained the address field value from the current instruction.
- ***Instruction Buffer Register (IBR)*** – This 40-bit register contained the most recently fetched instruction pair.
- ***Instruction Register (IR)*** – This 20-bit register held the instruction currently being executed.

The final progress report on the Princeton IAS machine as submitted by Goldstine, Pomerene, and Smith also presented a detailed description of the instruction set including opcodes (Goldstine, Pomerene, & Smith, 1954). This description is presented in Table 2-4.

Table 2-4 Princeton IAS Machine Instruction Set (Goldstine, Pomerene, & Smith, 1954)

Opcode (binary)	Operation description
11100101	Load AC with value from address in memory
11100100	Add value from address in memory to the AC
11101101	Load AC with negative of value from address in memory
11101100	Subtract value from address in memory from the AC
11110101	Load AC with absolute value of number from memory
11110100	Add absolute value of number from memory to the AC
11111101	Load AC with negative absolute value of number from memory
11111100	Subtract absolute value of number from memory from the AC
11100000	Multiply MQ with value from address in memory; put upper half of result in AC and lower half of result in MQ
11100010	Multiply MQ with value from address in memory; put only upper half of result in AC to round off lower 40 bits
11111000	Divide AC by value from address in memory; leave remainder in AC and put quotient in MQ
11100110	Load MQ with value from address in memory
11010100	Store AC to address in memory
11010101	Store AC to address in memory then clear AC
11010000	Unconditional jump to left instruction of destination address (step digit = 0)
11010000	Unconditional jump to right instruction of destination address (step digit = 1)
11010000	If value in AC is non-negative, jump to left instruction of destination address (step digit = 0)
11011000	If value in AC is non-negative, jump to right instruction of destination address (step digit = 1)
10100000	Shift AC right one position with LSB shifting into MQ
10100010	Shift AC right one position discarding LSB
10101000	Shift AC right one position discarding MSB
10100100	Transfer MQ to AC
N/A*	Replace address portion of left instruction at address in memory with left most 10 bits in AC
N/A*	Replace address portion of right instruction at address in memory with left most 10 bits in AC

* These instructions were part of the original IAS proposal, but do not appear as implemented in the final progress report

In addition to the instructions presented in Table 2-4, there were six I/O instructions for transferring data between memory and the outside world. Over the course of its life, the IAS machine used I/O including teletypewriters, punch cards, and magnetic wire (Ware, 1953).

2.6 What's Next?

These early architectures, though innovative, were tedious to program, due to their limited storage and simple instruction sets. The rest of this book will show how the Harvard and von Neumann architectures have been used as the foundation for developing more complex and powerful computers.

Chapter 3 will begin by showing how increased flexibility in memory addressing can expand the methods by which algorithms can be implemented in software. Chapter 4 will then show how to design an instruction set capable of handling any algorithms. This will then lead to a discussion of how instructions execute and how hardware can be designed for greater performance.

2.7 References

Bromley, A. G. (1998). Charles Babbage’s Analytical Engine, 1838. *IEEE Annals of the History of Computing , 20* (4), 29-45.

Burks, A. W., Goldstine, H. H., & von Neumann, J. (June 1946). *Preliminary discussion of the logical design of an electronic computing instrument.* Princeton, New Jersey: Institute for Advanced Study.

Burton, C. P. (1997). The Manchester University Small-Scale Experimental Machine Programmer's Reference Manual. *The Computer Conservation Society* (2).

Goldstine, H. H., Pomerene, J. H., & Smith, C. V. (1954). *Final Progress Report on the Physical Realization of an Electronic Computing Instrument.* Princeton, NJ: The Institute for Advanced Study - Electronic Computer Project.

Harvard University Computation Laboratory. (1946). *A Manual of Operation for the Automatic Sequence Controlled Calculator.* Cambridge, Massachusetts, USA: Harvard University Press.

IBM. (n.d.). *IBM's ASCC introduction (a.k.a. The Harvard Mark I).* Retrieved June 26, 2009, from IBM Archives: http://www-03.ibm.com/ibm/history/exhibits/markI/markI_intro.html

Menabrea, L. F., & Lovelace, A. (1843). *Sketch of the Analytical Engine Invented by Charles Babbage*. Retrieved June 16, 2009, from http://www.fourmilab.ch/babbage/sketch.html

Myhrvold, N., & Swade, D. (2008, May 1). An Evening with Nathan Myhrvold and Doron Swade: Discussing Charles Babbage's Difference Engine. Mountain View, California, USA: http://archive.computerhistory.org/lectures/2008.05.01-NathanMyhrvoldandDoronSwadeDiscussBabbagesDifferenceEngine.wmv.

Rojas, R. (1997). Konrad Zuse's Legacy: The Architecture of the Z1 and Z3. *IEEE Annals of the History of Computing , 19* (2), 5-16.

Smith, R. E. (1989). A Historical Overview of Computer Architecture. *IEEE Annals of the History of Computing , 10* (4), 277-303.

Swade, D. (2008). *The Engines - The Babbage Engines*. Retrieved May 11, 2009, from The Babbage Engine | Computer History Museum: http://www.computerhistory.org/babbage/engines/

von Neumann, J. (1945). First Draft of a Report on the EDVAC. *Reprinted in IEEE Annals of the History of Computing, 1993 , 15* (4), 27-75.

Ware, W. H. (1953). *This History and Development of the Electronic Computer Project at the Institute for Advanced Study.* Santa Monica, CA: RAND Corporation.

Zuse, K. (n.d.). *My first computer and first thoughts about data processing*. Retrieved May 13, 2009, from The History of Computing: http://ei.cs.vt.edu/~history/Zuse.html

Zuse, P. (2001). *The Life and Work of Konrad Zuse*. Retrieved May 11, 2009, from EPEMag Online, Wimborne Publishing: http://www.epemag.com/zuse/default.htm

2.8 Problems

1. Describe the benefits of separating code from data in an architecture's storage mechanism?
2. Describe the benefits of combining code and data in an architecture's storage mechanism?
3. Using the Z1 machine language presented in Table 2-1, write a program to compute $2x^2 + x - 5$ where x is a value stored in one of the ASCC's storage registers. Be sure to write the code as a

sequence of 8-bit instructions. In addition, identify the purpose of the referenced memory locations along with initial values if there are any.

4. Using the ASCC instruction set presented in Table 2-2, write a program to compute $2x^2 + x - 5$ where x is a value stored in one of the ASCC's storage registers. Be sure to write the code as a sequence of 24-bit instructions where a 1 represents a hole in the paper tape while a 0 represents the absence of a hole. In addition, identify the constant and storage registers to be used along with their initial values.
5. Using the SSEM instruction set presented in Table 2-3, write a program to compute *n!* (n-factorial) where *n* is an integer stored in the SSEM memory at location 30_{10}. The result is to be stored at memory location 31_{10}. Be sure to write the code as a sequence of 32-bit instructions and identify any memory locations used along with their initial values.
6. Using the IAS machine's instruction set presented in Table 2-4, write a program to compute *n!* (n factorial) where *n* is an integer stored in the SSEM memory at location 511_{10}. The result is to be stored at memory location 512_{10}. Be sure to write the code as a sequence of 40-bit instruction pairs and identify any memory locations used along with their initial values.
7. Using the programming language of your choice, create a simulator for the original SSEM computer. In your implementation, be sure to accurately match both the 32-bit 2's complement data format and the 32-bit instruction format as described in Figure 2-12. You do not need to simulate the format of placing the least significant bits to the left. Implement each of the opcodes defined in Table 2-3.

   At a minimum, you should have data elements simulating the following components of the IAS machine.

   - Using a 5-bit index into a 32-element array, simulate the 32-word memory.
   - The control register, CI, which contains the 5-bit address pointing to the next instruction to execute. Initialization of CI should force program to start at a specific address in memory, however, remember that CI is incremented immediately before

an instruction's execution.
- The accumulator, AC, which contains the 32-bit data being operated on.

In addition, be sure to implement the halt command which stops the machine. Upon encountering a HALT, have the simulator display the contents of its 1024-word memory in order to verify the results.

8. Using the programming language of your choice, create a simulator for the original IAS computer. In your implementation, be sure to accurately match both the 40-bit 2's complement data format and the 40-bit instruction pair format as described in Figure 2-15. Implement each of the opcodes defined in Table 2-4.

   At a minimum, you should have data elements simulating the following components of the IAS machine.

   - The 1024-word memory using a 10-bit index into a 1024-element array
   - The instruction register, IR, which contains the current 8-bit instruction.
   - The instruction buffer register, IBR, which contains the 8-bit "right" instruction that was stored while the "left" instruction was being executed.
   - The program counter, PC, which contains a pointer to the next instruction pair to execute. Initialization of PC should force program to start at a specific address in memory.
   - The accumulator, AC, which contains the 40-bit data being operated on.
   - The multiplier quotient, MQ, which contains the-40 bit data multiplier, lower half of multiplication result, or quotient.

   In addition, halt the machine if the step digit contains a zero. Upon encountering a HALT, have the simulator display the contents of the 1024-word memory in order to verify the results.