On Some Instructions Used In Cyber Security: Literature Based

Dr SRIDHAR SESHADRI
Ex Vice Chancellor and Professor Computer Science & Engineering
Email : drssridhar@yahoo.com

Abstract: **The Hack computer's instruction set architecture (ISA) and derived machine language is sparse compared to many other architectures. Although the 6 bits used to specify a computation by the ALU could allow for 64 distinct instructions, only 18 are officially implemented in the Hack computer's ISA. Since the Hack computer hardware has direct support for neither integer multiplication (and division) or function calls, there are no corresponding machine language instructions in the ISA for these operations. These instructions are discussed in this paper.**

*Keywords:***Hack Computer"S Instruction; Derived Machine Language; ALU; ISA; Integer Multiplication;**

INTRODUCTION

The Hack computer's instruction set architecture (ISA) and derived machine language is sparse compared to many other architectures. Although the 6 bits used to specify a computation by the ALU could allow for 64 distinct instructions, only 18 are officially implemented in the Hack computer's ISA. Since the Hack computer hardware has direct support for neither integer multiplication (and division) or function calls, there are no corresponding machine language instructions in the ISA for these operations.

Hack machine language has only two types of instructions, each encoded in 16 binary digits.

A-instructions

Instructions whose most significant bit is "0" are called A-instructions or address instructions. The A-instruction is bit-field encoded as follows:

$\mathbf{0}b_{14}b_{13}b_{12}b_{11}b_{10}b_9b_8b_7b_6b_5b_4b_3b_2b_1b_0$

0 – the most significant bit of a A-instruction is "0"

b_{14} - b_0 - these bits provide the binary representation of a non-negative integer in the decimal range 0 through 32767

When this instruction is executed, the remaining 15 bits are left-zero extended and loaded into the CPU's A-register. As a side-effect, the RAM register having the address represented by that value is enabled for subsequent read/write action in the next clock cycle.

C-instructions

The other instruction type, known as C-instructions (computation instructions), have "1" as the most significant bit. The remaining 15 bits are bit-field encoded to define the operands, computation performed, and storage location for the specified computation result. This instruction may also specify a program branch based on the most recent computation result.

The C-instruction is bit-field encoded as follows:

$\mathbf{1}x_1x_0ac_5c_4c_3c_2c_1c_0d_2d_1d_0j_2j_1j_0$

1 – the most significant bit of a C-instruction is "1"

x_1x_0 – these bits are ignored by the CPU and, by convention, are each always set to "1"

a – this bit specifies the source of the "y" operand of the ALU when it is used in a computation

c_0-c_5 – these six control bits specify the operands and computation to be performed by the ALU

d_2-d_0 – these three bits specify the destination(s) for storing the current ALU output

j_2-j_0 – these three bits specify an arithmetic branch condition, an unconditional branch (jump), or no branching

The Hack computer encoding scheme of the C-instruction is shown in the following tables.

In these tables,

- **A** represents the value currently contained in the A-register
- **D** represents the value currently contained in the D-register
- **M** represents the value currently contained in the data memory register whose address is contained in the A-register; that is, M == RAM[A]

Hack machine language computation function codes and assembly language mnemonics

a	c_5	c_4	c_3	c_2	c_1	c_0	ALU Output: f(x,y)	Mnemonic
0	1	0	1	0	1	0	Outputs 0; ignores all operands	0
0	1	1	1	1	1	1	Outputs 1; ignores all operands	1
0	1	1	1	0	1	0	Outputs -1; ignores all operands	-1
0	0	0	1	1	0	0	Outputs D; ignores A and M	D
0	1	1	0	0	0	0	Outputs A; ignores D and M	A
1	1	1	0	0	0	0	Outputs M; ignores D and A	M
0	0	0	1	1	0	1	Outputs bitwise negation of D; ignores A and M	!D
0	1	1	0	0	0	1	Outputs bitwise negation of A; ignores D and M	!A
1	1	1	0	0	0	1	Outputs bitwise negation of M; ignores D and A	!M
0	0	0	1	1	1	1	Outputs 2's complement negative of D; ignores A and M	-D
0	1	1	0	0	1	1	Outputs 2's complement negative of A; ignores D and M	-A
1	1	1	0	0	1	1	Outputs 2's complement negative of M; ignores D and A	-M
0	0	1	1	1	1	1	Outputs D + 1 (increments D); ignores A and M	D+1
0	1	1	0	1	1	1	Outputs A + 1 (increments A); ignores D and M	A+1
1	1	1	0	1	1	1	Outputs M + 1 (increments M); ignores D and A	M+1
0	0	0	1	1	1	0	Outputs D - 1 (decrements D); ignores A and M	D-1
0	1	1	0	0	1	0	Outputs A - 1 (decrements A); ignores D and M	A-1
1	1	1	0	0	1	0	Returns M-1 (decrements M); ignores D and A	M-1
0	0	0	0	0	1	0	Outputs D + A; ignores M	D+A
1	0	0	0	0	1	0	Outputs D + M; ignores A	D+M
0	0	1	0	0	1	1	Outputs D - A; ignores M	D-A
1	0	1	0	0	1	1	Outputs D - M; ignores A	D-M
0	0	0	0	1	1	1	Outputs A - D; ignores M	A-D
1	0	0	0	1	1	1	Outputs M - D; ignores A	M-D
0	0	0	0	0	0	0	Outputs bitwise logical And of D and A; ignores M	D&A
1	0	0	0	0	0	0	Outputs bitwise logical And of D and M; ignores A	D&M
0	0	1	0	1	0	1	Outputs bitwise logical Or of D and A; ignores M	D|A
1	0	1	0	1	0	1	Outputs bitwise logical Or of D and M; ignores A	D|M

Hack machine language computation result storage codes and assembly language mnemonics

d_2	d_1	d_0	Store ALU output in	Mnemonic
0	0	0	Output not stored	*none*
0	0	1	M	M
0	1	0	D	D
0	1	1	D and M	DM
1	0	0	A	A
1	0	1	A and M	AM

1	1	0	A and D	AD
1	1	1	A and D and M	ADM

Hack machine language branch condition codes and assembly language mnemonics

j_2	j_1	j_0	Branch if	Mnemonic
0	0	0	No branch	*none*
0	0	1	Output greater than 0	JGT
0	1	0	Output equals 0	JEQ
0	1	1	Output greater than or equal 0	JGE
1	0	0	Output less than	JLT
1	0	1	Output not equal 0	JNE
1	1	0	Output less than or equal 0	JLE
1	1	1	Unconditional branch	JMP

Assembly language

The Hack computer has a text-based assembly language to create programs for the hardware platform that implements the Hack computer ISA. Hack assembly language programs may be stored in text files having the file name extension ".asm". Hack assembly language source files are case sensitive. Each line of text contains one of the following elements:

- Blank line
- Comment
- Label declaration (with optional end-of-line comment)
- A-instruction (with optional end-of-line comment)
- C-instruction (with optional end-of-line comment)

Each of these line types has a specific syntax and may contain predefined or user defined symbols or numeric constants. Blank lines and comments are ignored by the assembler. Label declarations, A-instructions, and C-instructions, as defined below, may not include any internal white-space characters, although leading or trailing whitespace is permitted (and ignored).

Comments

Any text beginning with the two-character sequence "//" is a comment. Comments may appear on a source code line alone, or may also be placed at the end of any other program source line. All text following the comment identifier character sequence to end of line is completely ignored by the assembler; consequently, they produce no machine code.

Symbols and numeric constants

Hack assembly language allows the use of alphanumeric symbols for number of different specific purposes. A symbol may be any sequence of alphabetic (upper and lower case) or numeric digits. Symbols may also contain any of the following characters: under bar ("_"), period("."), dollar sign ("$"), and colon (":"). Symbols may not begin with a digit character. Symbols are case sensitive. User defined symbols are used to create variable names and labels (see below).

The Hack assembly language assembler recognizes some predefined symbols for use in assembly language programs. The symbols R0, R1, …, R15 are bound respectively to the integers 0 through 15. These symbols are meant to represent general purpose registers and the symbols values therefore represent data memory addresses 0 through 15. Predefined symbols SCREEN and KBD are also specified to represent the data memory address of the start of memory-mapped virtual screen output (16384) and keyboard input (24756). There are a few other symbols (SP, LCL, ARG, THIS, and THAT) that are used in building the operating system software stack.

A sting of decimal (0-9) digits may be used to represent a non-negative, decimal constant in the range 0 through 32,767. The use of the minus sign to indicate a negative number is not allowed. Binary or octal representation is not supported.

Variables

User defined symbols may be created in an assembly language program to represent variables; that is, a named RAM register. The symbol is bound at assembly to a RAM address chosen by the assembler. Therefore, variables must be treated as addresses when appearing in assembly language

source code. Variables are implicitly defined in assembly language source code when they are first referenced in an A-instruction. When the source code is processed by the assembler, the variable symbol is bound to a unique positive integer value in beginning at address 16. Addresses are sequentially bound to variable symbols in the order of their first appearance in the source code. By convention, user-defined symbols that identify program variables are written in all lower case.

Labels

Labels are symbols delimited by left "(" and right ")" parenthesis. They are defined on a separate source program line and are bound by the assembler to the address of the instruction memory location of the next instruction in the source code. Labels may be defined only once, but they may be used multiple times anywhere within the program, even before the line on which they are defined. By convention, labels are expressed in all-caps. They are used to identify the target address of branch C-instructions.

A-instructions

The A-instruction has the syntax "*@xxxx*", where *xxxx* is either a numeric decimal constant in the range 0 through 32767, a label, or a variable (predefined or user defined). When executed, this instruction sets the value of the A register and the M pseudo-register to a 15-bit binary value represented by "*xxxx*". The 15-bit value is left-zero extended to 16-bits in the A register.

The A-instruction may be used for one of three purposes. It is the only means to introduce a (non-negative) numeric value into the computer under program control; that is, it may be used to create program constants. Secondly, it is used to specify a RAM memory location using the M pseudo-register mechanism for subsequent reference by a C-instruction. Finally, a C-instruction which specifies a branch uses the current value of the A register as the branch target address. The A-instruction is used to set that target address prior to the branch instruction, usually by reference to a label.

C-Instructions

C-instructions direct the ALU computation engine and program flow control capabilities of the Hack computer. The instruction syntax is defined by three fields, referred to as "comp", "dest", and "jump". The comp field is required in every C-instruction. The C-instruction syntax is "dest=comp**;**jump". The "=" and ";" characters are used to delimit the fields of the instruction. If the dest field is not used, the "=" character is omitted. If the jump field is not used, the ";" character is omitted. The C-instruction allows no internal spaces.

The *comp* field must be one of the 28 documented mnemonic codes defined in the table above. These codes are considered distinct units; they must be expressed in all-caps with no internal spaces. It is noted that the 6 ALU control bits could potentially specify 64 computational functions; however, only the 18 presented in the table are officially documented for recognition by the assembler.

The *dest* field may be used to specify one or more locations to store the result of the specified computation. If this field is omitted, along with the "=" delimiter, the computed value is not stored. The allowed storage location combinations are specified by the mnemonic codes defined in the table above.

The jump field may be used to specify the address in ROM of the next instruction to be executed. If the field is omitted, along with the ";" delimiter, execution continues with the instruction immediately following the current instruction. The branch address target, in ROM, is provided by the current value of the A register if the specified branch condition is satisfied. If the branch condition fails, execution continues with the next instruction in ROM. Mnemonic codes are provided for six different comparisons based on the value of the current computation. Additionally, an unconditional branch is provided as a seventh option. Because the comp field must always be supplied, even though the value is not required for the unconditional branch, the syntax of this instruction is given as "0;JMP". The branch conditions supported are specified in the table above.

Assembler

Freely available software supporting the Hack computer includes a command line assembler application. The assembler reads Hack assembly language source tiles (*.asm) and produces Hack machine language output files (*.hack). The machine language file is also a text file. Each line of this file is a 16-character string of binary digits that represents the encoding of each corresponding executable line of the source text file according to the specification described in the section "Instruction set architecture (ISA) and machine language". The file created may be loaded into the Hack computer emulator by a facility provided by the emulator user interface.

Example Assembly Language Program

Following is an annotated example program written in Hack assembly language. This program sums the first 100 consecutive integers and places the result of the calculation in a user-defined variable called "sum". It implements a "while" loop construct to iterate though the integer values 1 through 100 and adds each integer to a "sum" variable. The user-defined variable "cnt" maintains the current integer value through the loop. This program illustrates all of the features of the "documented" assembly language capabilities of Hack Computer except memory-mapped I/O. The contents of the Hack assembly language source file are shown in the second column in bold font. Line numbers are provided for reference in the following discussion but do not appear in the source code. The Hack machine code produced by the assembler is shown in the last column with the assigned ROM address in the preceding column. Note that full-line comments, blank lines, and label definition statements generate no machine language code. Also, the comments provided at the end of each line containing an assembly language instruction are ignored by the assembler.

The assembler output, shown in the last column, is a text string of 16 binary characters, not 16-bit binary integer representation.

Line Nbr	Hack Assembly Language Program	Operating Notes	Instruction Type	ROM Addr	Hack Machine Code
01	**// Add consecutive integers 1 thru 100**	Comment that describes program action	*Full-line comment*	- - - -	*No code generated*
02	**// sum = 1 + 2 + 3 + ... + 99 + 100**	Comments are ignored by assembler	*Full-line comment*	- - - -	*No code generated*
03		Blank source lines are ignored by assembler	*Blank line*	- - - -	*No code generated*
04	**@cnt // loop counter declaration**	Variable symbol "cnt" bound to 16	A-instruction	00	0000000000001000
05	**M=1 // initialize loop counter to 1**	RAM[16] ← 1	C-instruction	01	1110111111001000
06	**@sum // sum accumulator declaration**	Variable symbol "sum" bound to 17	A-instruction	02	0000000000010001
07	**M=0 // initialize sum to 0**	RAM[17] ← 0	C-instruction	03	1110101010001000
08	**(LOOP) // start of while loop**	Label symbol bound to ROM address 04	*Label declaration*	- - - -	*No code generated*
09	**@cnt // reference addr of cnt**	M ← 16	A-instruction	04	0000000000001000
10	**D=M // move current cnt value to D**	D ← RAM[16]	C-instruction	05	1111110000010000
11	**@100 // load loop limit into A**	A ← 100	A-instruction	06	0000000001100100
12	**D=D-A // perform loop test computation**	D ← D – A	C-instruction	07	1110010011010000
13	**@END // load target destination for branch**	M ← 18	A-instruction	08	0000000000010010

14	**D;JGT //exit loop if D > 0**	Conditional branch	C-instruction	09	1110001100000001
15	**@cnt // reference addr of cnt**	M ← 16	A-instruction	10	0000000000001000
16	**D=M // move current cnt value to D**	D ← RAM[16]	C-instruction	11	1111110000010000
17	**@sum // reference address of sum**	M ← 17	A-instruction	12	0000000000010001
18	**M=D+M // add cnt to sum**	M ← D + RAM[17]	C-instruction	13	1111000010001000
19	**@cnt // reference addr of cnt**	M ← 16	A-instruction	14	0000000000001000
20	**M=M+1 // increment counter**	RAM[16] ← RAM[16] + 1	C-instruction	15	1111110111001000
21	**@LOOP // load target destination for branch**	M ← 4	A-instruction	16	0000000000000100
22	**0;JMP // jump to LOOP entry**	Unconditional branch	C-instruction	17	1110101010000111
23	**(END) // start of terminating loop**	Label symbol bound to ROM address 18	*Label declaration*	- - - -	*No code generated*
24	**@END // load target destination for branch**	M ← 18	A-instruction	18	0000000000010010
25	**0;JMP // jump to END entry**	Unconditional branch	C-instruction	19	1110101010000111

Note that the instruction sequence follows the pattern of A-instruction, C-instruction, A-instruction, C-instruction, ... This is typical for Hack assembly language programs. The A-instruction specifies a constant or memory address that is used in the subsequent C-instruction. All three variations of the A-instruction are illustrated. In line 11 (@100), the constant value 100 is loaded into the A register. This value is used in line 12 (D=D-A) to compute the value used to test the loop branch condition. Since line 4 (@cnt) contains the first appearance of the user-defined variable "cnt", this statement binds the symbol to the next unused RAM address. In this instance, the address is 16, and that value is loaded into the A register. Also, the M pseudo-register also now references this address, and RAM[16] is made the active RAM memory location.

The third use of the A-instruction is seen in line 21 (@LOOP). Here the instruction loads the bound label value, representing an address in ROM memory, into the A register and M pseudo-register. The subsequent unconditional branch instruction in line 22 (0;JMP) loads the M register value into the CPU's program counter register to effect control transfer to the beginning of the loop. The Hack computer provides no machine language instruction to halt program execution. The final two lines of the program (@END and 0;JMP) create an infinite loop condition which Hack assembly programs conventionally use to terminate programs designed to run in the CPU emulator.

REFERENCES

[1]. Liu, Jinan; Rahman, Sajjadur; Serletis, Apostolos (2020). "Cryptocurrency Shocks". SSRN Electronic Journal. doi:10.2139/ssrn.3744260. ISSN 1556-5068. S2CID 233751995.

[2]. Malvino, Albert P., & Brown, Jerald A. (1993). Digital Computer Electronics, 3rd Edition. New York, New York: Glencoe McGraw-Hill

[3]. Matteo D'Agnolo. "All you need to know about Bitcoin". timesofindia-economictimes. Archived from the original on 26 October 2015.

[4]. Michael G. Noblett; Mark M. Pollitt; Lawrence A. Presley (October 2000). "Recovering and examining computer forensic evidence". Retrieved 26 July 2010.

[5]. Millman, Renee (15 December 2017). "New polymorphic malware evades three-quarters of AV scanners". SC Magazine UK.

[6]. Milutinović, Monia (2018). "Cryptocurrency". Ekonomika. **64** (1): 105–122. doi:10.5937/ekonomika1801105M. ISSN 0350-137X.

[7]. Moore, R. (2005) "Cyber crime: Investigating High-Technology Computer Crime," Cleveland, Mississippi: Anderson Publishing.

[8]. Nakashima, Ellen (26 January 2008). "Bush Order Expands Network Monitoring: Intelligence Agencies to Track Intrusions". The Washington Post. Retrieved 8 February 2021.

[9]. Null, Linda, & Lobur, Julia. (2019). The Essentials of Computer Organization and Architecture. 5th Edition. Burlington, Massachusetts: Jones and Bartlett Learning.

[10]. Pagliery, Jose (2014). Bitcoin: And the Future of Money. Triumph Books. ISBN 978-1629370361. Archived from the original on 21 January 2018. Retrieved 20 January 2018.

[11]. Parker D (1983) Fighting Computer Crime, U.S.: Charles Scribner's Sons.

[12]. Patt, Yale N., & Patel, Sanjay J. (2020). Introduction to Computing Systems: From Bits and Gates to C and Beyond, 3rd Edition. New York, New York: McGraw Hill Education.

[13]. Pernice, Ingolf G. A.; Scott, Brett (20 May 2021). "Cryptocurrency". Internet Policy Review. **10** (2). doi:10.14763/2021.2.1561. ISSN 2197-6775.

[14]. Perrin, Chad (30 June 2008). "The CIA Triad". techrepublic.com. Retrieved 31 May 2012.

[15]. Petzold, Charles. (2009). Code: The Hidden Language of Computer Hardware and Software. Redmond, Washington: Microsoft Press.

[16]. Pitta, Julie. "Requiem for a Bright Idea". Forbes. Archived from the original on 30 August 2017. Retrieved 11 January 2018.

[17]. Polansek, Tom (2 May 2016). "CME, ICE prepare pricing data that could boost bitcoin". Reuters. Retrieved 3 May 2016.

[18]. Ruddenklau, Ian Pollari,Anton (August 9, 2021). "Pulse of Fintech H1 2021 – Global - KPMG Global". KPMG. Retrieved January 3, 2022.

[19]. Sanicola, Lenny (February 13, 2017). "What is FinTech?". Huffington Post. Retrieved August 20, 2017.

[20]. Schatz, Daniel; Bashroush, Rabih; Wall, Julie (2017). "Towards a More Representative Definition of Cyber Security". Journal of Digital Forensics, Security and Law. **12** (2). ISSN 1558-7215.

[21]. Schueffel, Patrick (March 9, 2017). "Taming the Beast: A Scientific Definition of Fintech". Journal of Innovation Management. **4** (4): 32–54. doi:10.24840/2183-0606_004.004_0004.

[22]. Scott, John Clark. (2009). But How Do It Know? The Basic Principles of Computers for Everyone. Oldsmar, Florida: John C. Scott.

[23]. Stevens, Tim (11 June 2018). "Global Cybersecurity: New Directions in Theory and Methods" (PDF). Politics and Governance. **6** (2): 1–4. doi:10.17645/pag.v6i2.1569.

[24]. Stoneburner, G.; Hayden, C.; Feringa, A. (2004). "Engineering Principles for Information Technology Security" (PDF). csrc.nist.gov. doi:10.6028/NIST.SP.800-27rA.

[25]. Van Loo, Rory (February 1, 2018). "Making Innovation More Competitive: The Case of Fintech". UCLA Law Review. **65** (1): 232.

[26]. Warren G. Kruse, Jay G. Heiser (2002). Computer forensics: incident response essentials. Addison-Wesley. p. 392. ISBN 978-0-201-70719-9.

[27]. Warren G. Kruse; Jay G. Heiser (2002). Computer forensics: incident response essentials. Addison-Wesley. p. 392. ISBN 978-0-201-70719-9. Retrieved 6 December 2010.

[28]. Webroot (24 July 2018). "Multi-Vector Attacks Demand Multi-Vector Protection". MSSP Alert. Retrieved 11 May 2022.

[29]. Yaffe-Bellany, David (15 September 2022). "Crypto's Long-Awaited 'Merge' Reaches the Finish Line". The New York Times. Retrieved 16 September 2022.

[30]. Yasinsac, A.; Erbacher, R.F.; Marks, D.G.; Pollitt, M.M.; Sommer, P.M. (July 2003). "Computer forensics education". IEEE Security & Privacy. **1** (4): 15–23. doi:10.1109/MSECP.2003.1219052.

