Machine and Assembly Language *

Jens Lechtenbörger

IT Systems, Summer Term 2024

This presentation deals with the machine language for the Hack computer. As the machine language contains instructions of 0s and 1s that are hard to read and write for human beings, we actually start with assembly language, a symbolic representation of machine language.

1 Introduction

Let us begin with a look at machine languages in general.

1.1 Machine Languages

- Duality
  - **Machine language** (= instruction set) as **abstract description** of computer architecture
    - Instructions as sequences of 0s and 1s
  - Hardware as physical means for realizing abstract machine language

Two forms of duality exist in the realm of machine language. First, machine language serves as abstract description of the computer architecture. In machine language, instructions are represented as sequences of 0s and 1s, forming the fundamental code that directs the behavior of a computer. The physical hardware then is supposed to implement the abstract machine language.

- Another duality
  - Machine language
    - Sequences of 0s and 1s
  - **Assembly language** (also called assembler)
    - **Mnemonic** notation (easier to remember, read, write for human beings)
    - Software called **assembler** translates assembly code to machine code

Another duality relates to the representation of instructions. While machine language relies on binary sequences, which can be complex and challenging for humans to work with directly, assembly language offers a more human readable alternative. Assembly language, also known as assembler, uses mnemonic notation, which is easier to remember, read, and write for human beings. This notation simplifies the process of programming by providing symbolic, or mnemonic, representations of machine instructions. Assembler software translates assembly code into machine code, allowing computers to execute the instructions written by programmers at a higher level of abstraction.

*This PDF document is an inferior version of an OER HTML page; free/libre Org mode source repository.

1.2 Today's Core Question

- How can a machine language for a computer embedding the Hack ALU look like?
  - Based on Chapter 4 of (Nisan and Schocken 2005)

The core topic of this presentation is the specification of a machine language for the Hack computer, which in particular enables computations of the Hack ALU.

1.3 Learning Objectives

- Read and write programs in Hack assembly language

After working through this presentation you should be able to read and write programs in the Hack assembly language.

1.4 Retrieval Practice

- What is the fetch-execute-decode cycle?
- What are the core parts of the Hack computer architecture?
  - Which registers does it contain?

Please take a brief break and write down answers to these question, without using previous class material.

Agenda

The agenda of this presentation is as follows. After this introduction, we revisit the target Hack computer with more details. Afterwards, we define the assembly and machine language for our computer.

2 Aspects of the Hack Computer

Let us revisit, and specify in more detail, the core parts that the Hack computer is supposed to contain. Clearly, those parts need to be manageable by the machine language.

2.1 Hack Computer

- Hack computer

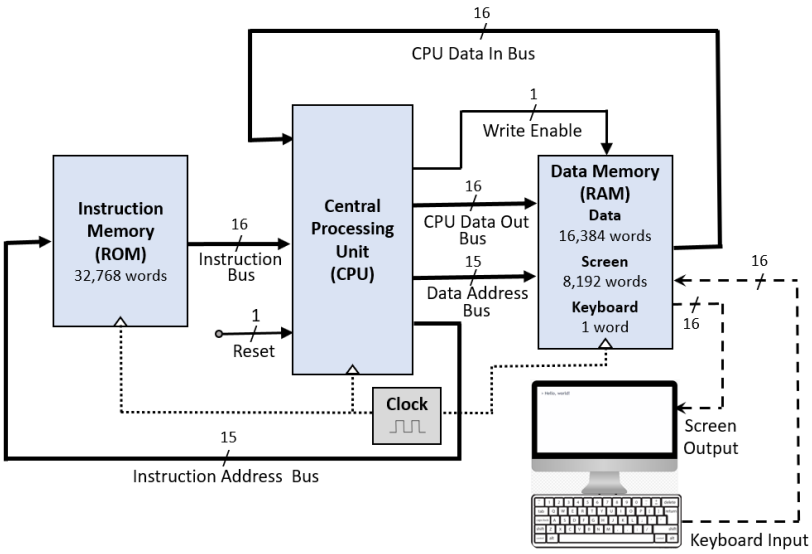

Figure 1: "Hack Computer" by user52174 under CC BY-SA 4.0; from StackExchange

- Major components
  - CPU with registers A, D, PC
  - RAM, ROM
  - Input/Output (I/O) devices
    - Keyboard and screen
    - **Memory maps** for interaction and abstraction
  - Buses for connections
    - Addresses for data memory and instruction memory with 15 bits

This schematic of the Hack computer shows its major components and their connections in terms of buses.

As in an earlier sketch, our target computer contains data memory with RAM and instruction memory with ROM. Now, the connections between CPU and memory are shown explicitly as buses. Note that CPU and RAM are connected by three buses. First, the address bus transfers the address of the memory location which should be read or written by the CPU. Second, another bus transfers data from RAM into the CPU in response to read requests. The third bus transfers data for writing by the CPU into RAM.

Although the inner workings of the CPU are not shown here, it is important to remember that it contains a specific ALU and three registers, namely A-register, D-register, and program counter. These design decisions have a large impact on the machine language. Clearly, there should be machine instructions for each of the ALU operations, and the instructions need to specify on what operands the ALU is supposed to operate.

Also note that the instruction address bus between CPU and ROM carries the value from the program counter. In response, the ROM emits the next instruction to execute by the CPU via the instruction bus.

As a new aspect we see a screen as output device and a keyboard as input device. Importantly, as explained subsequently, both I/O devices are mapped into data memory: The screen is mapped with a range of 8192 words that represent pixels on screen. The keyboard is mapped to a single address whose value represents the currently pressed key; thus, its value changes automatically in response to key presses, with standardized codes for each key. Note that this form of memory mapped I/O abstracts away all implementation details of the concrete hardware devices.

2.2 I/O Handling: Screen

- Programmer interacts with screen and keyboard through **memory maps**
- **Screen**, output device
  - Black-and-white, 512 x 256 pixels, 1 bit per pixel
    - Pixels represented by binary values in memory map for screen
      - $\frac{512 \cdot 256}{16} = 8192$ words

In Hack, programmers interact with screen and keyboard through so-called memory maps.

The Hack screen has a resolution of 512 columns and 256 rows, where each pixel can be black or white. Thus, a single bit can represent the state of a pixel, and all pixels together are represented by 8192 words.

The screen is an output device, to which programmers can send data but which does not produce data on its own.

    - Memory map starting at address 16384
      - Each row with 512 pixels represented by $\frac{512}{16} = 32$ consecutive 16-bit words
      - Least significant bit of each word represents left-most pixel

In Hack, the first 16384 words of data memory are implemented by a RAM chip and can be used freely by programmers. The subsequent addresses have the following special interpretation: They are used as memory area, or memory map, for the 8192 words representing pixels on screen. Thus, if one writes into that memory map, pixels on screen are turned on or off.

Apparently, for each row of 512 pixels, 32 words are necessary, and by convention the least significant bit of each word represents the left-most pixel. You can try this out in the CPU Emulator by writing values into the address range of the screen's memory map.

2.3 I/O Handling: Keyboard

- Programmer interacts with screen and keyboard through **memory maps**
  - **Keyboard**, input device
    - Single word memory map at memory address 24576
    - Listen to keyboard: Read memory at address 24576
      - When key pressed: 16-bit ASCII code appears at 24576 (in addition to special codes, e.g., newline = 128)
      - When no key pressed: value 0 at address 24576

The Hack keyboard is also a memory mapped device, in this case an input device, from which programmers can receive data. When a key is pressed, the keyboard generates a code representing that key and writes that code to data memory at address 24576. When no key is pressed, that location contains the value 0.

You can try this out in the CPU Emulator by pressing a key and watching the keyboard's memory location change.

2.4 CPU Emulator

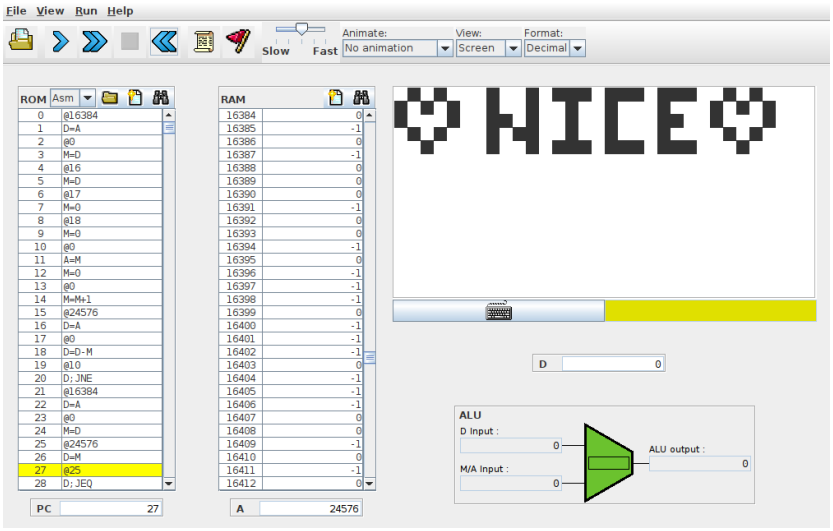

Figure 2: ("Screenshot of CPU Simulator for Nand to Tetris" under GPLv2; from GitLab)

- Screenshot for student's program
  - Note
    - Registers A, D, PC
    - Assembly code in ROM
    - Screen and its memory map in RAM

You can experiment with the Hack computer in the CPU Emulator of Nand to Tetris, of which you see a screenshot here. Importantly, you can load programs written in assembly language, which are then shown as ROM contents. Upon execution, you can watch changes in registers, in RAM, and on screen, and you can interact via the keyboard.

Note that this emulator provides considerable flexibility over real ROM chips: You can type single machine instructions directly into ROM locations and execute them for testing purposes. Also, you can write values into any register, including the program counter. These features are highly useful for experiments and learning.

On a side note, the program running here was written by a student and allows to print characters to screen, controlled by the keyboard. Source code is available on GitHub.

2.4.1 Video of CPU Emulator

- Two routes are possible from here
  - Either you continue with this presentation for an explanation of the assembly language
  - Or you pause here and watch a video explaining major assembly language aspects using the CPU Emulator in Learnweb

At this point you may continue in one of two ways. Either you continue with this presentation or you head over to Learnweb for a video. That video uses the CPU Emulator to explain a sample program along with major aspects of the assembly language. If you find the next slides too abstract, maybe go to the video and come back afterwards.

3 Hack Assembly Language

Let us look at the assembly language for the Hack computer, given the target architecture sketched so far.

3.1 Memory Access via A Register

- Hack is 16-bit machine
  - 16 bits for words, registers, machine instructions
    - 6 bits for ALU operations, 15 bits for memory addresses
    - Thus, operation and address do not fit into one instruction

As design decision for Hack, words have a size of 16 bits, which is also the size for registers and machine instructions. Then, you already know that the ALU features 6 control bits.

Quite likely, at least some instructions contain these control bits to initiate ALU operations. If those operations are supposed to work on a value in data memory, then we need to manage the address of that value somewhere. For data memory, which includes `RAM16K` and memory maps for screen and keyboard, we need addresses with a size of 15 bits. As a result, ALU operation and address do not fit into one instruction.

- Hack uses implicit location "M" for memory access
  - "Operations" involving memory locations require 2 instructions
    1. Specify location "M"
    2. Operate on that location

To solve this puzzle, Hack uses an implicit location M for memory access, and operations involving memory locations are implemented with multiple machine instructions: First, one specifies the address for M. Afterwards, other machine instructions can refer to M, which implicitly refers to the address specified earlier.

  - Convention: **M** refers to memory word whose **address** is in **A**
    - In other words, M means Memory[A]
      - Therefore, A can be understood as **address register**

    In Hack, the A-register serves as **address register** in the following sense: If we load a value into that register, then subsequently M refers to that word in data memory whose address is stored in the A-register.

    - Example: "D = Memory[42] + 1" (D is useful as **data register**)
      1. Load 42 into A (A-instruction)
      2. Execute D = M + 1 (C-instruction)

You see an example here, which increments the value stored in data memory at address 42 and assigns the result to the D-register: First, load 42 into the A-register, then execute the increment operation with reference to M.

As a side note, the D-register can be understood as general-purpose **data register**.

3.2 A-Instruction

- Simple syntax: `@value`
  - `value` is a non-negative number (or a symbol)
    - E.g., `@42`

  - Effect: Write `value` into A register
    * Thus, `@42` writes 42 into A register

More precisely, we use the so-called A-instructions to load values into the A-register. As shown here, an A-instruction starts with the At sign and provides a value. That value is then written into the A-register.

3.2.1 A-Instructions in 2 Steps

- 2 steps as pairs of A-instruction and C-instruction (try out in CPU Emulator)

  As mentioned earlier, we usually combine an A-instruction with subsequent instructions.

  - Assign constant to D register

    ```
    @42  // Write 42 into A register.
    D=A  // Copy contents of A register into D register.
    ```

    As first example, you see here how to assign a constant to the D-register.
    Note that in this case the A-register is **not** used for addressing.

  - Access RAM[42]

    ```
    @42  // Write 42 into A register.
    D=M  // M refers to RAM[A], which is RAM[42].
     // Thus, copy contents of RAM[42] to D.
    ```

    The second example shows how to copy the contents of RAM location 42 into the D-register.

  - Access ROM[42]

    ```
    @42  // Write 42 into A register.
    JMP  // JMP loads the PC with the value of the A register.
     // Thus, execution continues at ROM[42].
    ```

    Our machine language also knows so-called jump instructions, which are like goto statements of other languages. They allow continuing execution at some chosen target location. Also in this case, the A-register is used implicitly to specify the target address. This time, the address is used in ROM, as instructions are stored in ROM.

3.3 C-Instruction (Excerpt of Full Syntax)

- On previous slide, `D=A` and `D=M` are C-Instructions

  Let us now look at a subset of C-instructions, which involve assignments, for which the previous slide provided simple examples.

- Assignments, more generally (excerpt of arithmetic calculations)
  - `x` in {`A`, `D`, `M`}, `y` in {`A`, `D`, `M`, `1`}
  - `dest` in {`null`, `M`, `D`, `MD`, `A`, `AM`, `AD`, `AMD`}
    - `dest = x+y`
      - E.g., `D=D+A`, `MD=M+1`
    - `dest = x-y`
    - `dest = x`
    - `dest = 0`
    - `dest = 1`
    - `dest = -1`

Two issues:

- What to compute?
- Where to store the result?

An assignment specifies two aspects: First, the right-hand side defines what to compute. Second, the left-hand side states where to store the result.

More specifically, on the right-hand side we can not only use simple registers, but also selected constants and ALU operations involving operands. As shown here, the first operand, denoted x, may be the A-register, the D-register, or the implicit memory location M. The second operand, denoted y, may in addition also be chosen as constant 1.

As destinations, we can use any subset of A-register, D-register, and implicit memory location.

As computation on the right-hand side, any ALU operation is possible. Only a subset is shown.

3.4 Self-Study: Coding Exercise

- Program using Hack instructions:
  - Set A to 17
  - Set D to A-1
  - Set both A and D to A + 1
  - Set D to 19
  - Set RAM[5034] to D - 1
  - Set RAM[53] to 171
  - Load RAM[7], add 1, and store the result in D
  - Increment the number stored in the RAM location whose address is stored in RAM[42]

If you get stuck, see here for solutions.
Please take a break and solve the tasks shown here.

3.5 C-Instruction

```
dest=comp;jump // comp is mandatory; dest and jump are optional
```

- Where
  - `comp` is one of

```
0,1,-1,D,A,!D,!A,-D,-A,D+1,A+1,D-1,A-1,D+A,D-A,A-D,D&A,D|A,
       M,   !M,   -M,    M+1,    M-1,D+M,D-M,M-D,D&M,D|M
```

- `dest` is one of

  `null, M, D, MD, A, AM, AD, AMD`

- `jump` is one of

  `null, JGT, JEQ, JGE, JLT, JNE, JLE, JMP`

The full syntax of C-instructions is documented here. A computation by the ALU is mandatory, even if it can be as simple as computing a 0.
In addition, the computed value can be assigned to a destination.
Finally, the instruction can specify a jump, which is explained subsequently.

3.5.1 (Conditional) Jumping

- Jump loads the program counter (PC) with new location
  - Used for `if` and `goto` statements (e.g., jump to `else` part)

Jump instructions load a new ROM address into the program counter. Then, execution does not continue with the next instruction, but continues at a chosen target location. Thus, jump instructions are similar to `goto` statements of other languages, and they are also used to implement `if` statements.

- By **convention**, use only 2 types of jumps
  - **Unconditional jump**: `0;JMP`
    - Thus, `dest` is `null`, comp is `0`
    - Execution continues at ROM[A]

By convention, we use only 2 types of jump instructions.

First, to jump unconditionally, i.e., to jump in any case, the instruction shown here is used. This instruction lacks the optional destination part of C-instructions, computes 0 for the mandatory computation part (ignoring the result), and performs a jump.

The A-register specifies the jump target, which means that the next instruction to be executed is the one whose address is contained in the A-register.

  - **Conditional jumps** refer to D register: `D;JGT`, `D;JEQ`, ...
    - Thus, `dest` is `null`, comp is `D`
      Second, jumps can be tied to conditions. Again, this instruction lacks the optional destination part of C-instructions. This time, however, the D-register determines whether the jump takes effect or whether the next instruction is executed. Again, using the D-register is a convention, as any computation could be executed by the ALU and used to check the condition.
    - `D;JGT` jumps if value of D register is Greater Than 0
      Here you see the syntax for a jump that takes place if the value of the D-register is greater than 0.
      If that value is 0 or negative, the program counter is just incremented to execute the next instruction.
    - Similarly for `D;JEQ` (if EQual to 0); `D;JGE` (if Greater or Equal to 0), `D;JLT` (if Less Than 0), `D;JNE` (if Not Equal to 0), `D;JLE` (if Less than or Equal to 0)
      More variants of conditional jumps are available that test for other conditions.

3.5.2 Example: Max of 2 Numbers

```
// Write the larger of the two variables x and y to RAM[0].
// Idea: Compute y-x.  If result is >=0, then y is larger.
  @x    // Load location of x to A.  Replaced by @16 in CPU Emulator.
  D=M   // Load value of x to D.
  @y    // Load location of y to A.  Replaced by @17.
  D=M-D // Compute y-D (=y-x) to D.
  @Y_IS_LARGER // Load Jump target to A.  Replaced by @10.
  D;JGE        // If D is GE (greater or equal to 0), Jump.
       // Otherwise, continue with next instruction.
  // If we are here, x is larger.
  // Load x to D and jump elsewhere for writing.
  @x
  D=M
  @STORE_RESULT // Load Jump target to A.  Replaced by @12.
  0;JMP         // Jump unconditionally.
(Y_IS_LARGER)   // Declare Jump target.
// Next instruction (@y) is in ROM[10].
// @Y_IS_LARGER is the address of that instruction.
// If we are here, y is larger.  Copy via D to RAM[0].
  @y
  D=M
(STORE_RESULT)  // Declare Jump target.
// Next two instructions write D to RAM[0].
  @0
  M=D
(END)           // Declare Jump target for end of program (ROM[14]).
// By convention, infinite loops end programs in Hack.
  @END  // Load Jump target to A.
  0;JMP // Jump unconditionally to previous A-instruction.
```

(Source file)

This program computes the maximum of the two variables x and y. It stores the maximum in RAM location 0.

Clearly, we need to compare two values against each other. However, our ALU does not allow doing so directly. Instead, we can subtract one value from the other and compare the results to 0.

In the beginning, we use a pattern seen earlier to load x into the D-register. Recall the use of A-register and implicit memory location M.

Afterwards, we subtract the contents of the D-register from y and assign the result to the D-register.

Now we can compare the result against 0 in a conditional jump. Note that we use a label as jump target here, which is defined further below in line 16.

If the condition is not satisfied, x is larger than y. Thus, we load x into the D-register and jump to a place where the value of the D-register is stored into RAM location 0. Again, we use a label to specify the jump target. This time, we jump unconditionally.

Now we deal with the case where y is larger. Here, we load y into the D-register.

Next, we store the contents of the D-register to RAM location 0. Note that we use these instructions in any case, either from the conditional jump for x or in the sequence of instructions for y.

At this point, the D-register contains the maximum of x and y. Thus, our program is complete.

In Hack, we follow the convention to indicate the end of a program by an infinite loop.

Thus, if you are asked to write a Hack assembly program, you can always write down these lines towards the end of your program without thinking.

Please try this out in the CPU emulator

3.5.3 Symbols

- Symbol classes

  To simplify programming, assembly languages permit the use of certain classes of symbols.

  - **Predefined**
    - @SCREEN is the same as @16384 (first address of screen memory map)
    - @KDB is the same as @24576 (keyboard memory location)
    - (And more, not important for us)

    First, some predefined symbols exist. One can use these names instead of the addresses for memory maps of screen and keyboard in A-instructions.

  - **Labels** (in `UPPER` case by convention)
    - Destinations for `JMP` commands
    - Declared inside parentheses
      - E.g., `(Y_IS_LARGER)` and `(END)` above
      - Used without parentheses in A-instruction, e.g., `@END`

    Importantly, in the context of jump commands, we need to use A-instructions with the addresses of jump targets as ROM locations. While we can count the instructions and come up with the correct addresses ourselves, it is more convenient to use labels.

    You saw examples for labels above. Importantly, such labels do not specify instructions. Instead, they just provide references for ROM locations. The CPU Emulator or an assembler translates all real instructions and can then determine the address of the next instruction following a label. This address is then used as value for the label. Actually, when you load the above program into the CPU Emulator you see that no symbols occur at all as they have been replaced with their values already.

  - **Variables** (in `lower` case by convention)
    - Other symbols are treated as variables
      - E.g., `x` and `y` above
      - Translated to unique memory addresses by assembler and CPU Emulator, starting at RAM[16]

    Other symbols are treated as variables. By convention, variables occurring in a program are mapped to RAM locations starting at address 16. E.g., above each occurrence of variable x is replaced with address 16.

4 Hack Machine Language

Let us now see how to represent Hack instructions as sequences of 0s and 1s.

4.1 A-Instruction

- Assembler: `@value`
- Binary: 16-bit number starting with 0
  - E.g., `@42` → `0000000000101010`

Mapping A-instructions to binary is quite simple. We use a fixed leading bit of 0, followed by the binary representation of the value.

4.2 C-Instruction

- C-instruction in assembler: `dest=comp;jump`
- C-instruction in binary as 16-bit number starting with three 1s:

`1 1 1 a c1 c2 c3 c4 c5 c6 d1 d2 d3 j1 j2 j3`

(when a=0) comp	c1	c2	c3	c4	c5	c6	(when a=1) comp
0	1	0	1	0	1	0	
1	1	1	1	1	1	1	
-1	1	1	1	0	1	0	
D	0	0	1	1	0	0	
A	1	1	0	0	0	0	M
!D	0	0	1	1	0	1	
!A	1	1	0	0	0	1	!M
-D	0	0	1	1	1	1	
-A	1	1	0	0	1	1	-M
D+1	0	1	1	1	1	1	
A+1	1	1	0	1	1	1	M+1
D-1	0	0	1	1	1	0	
A-1	1	1	0	0	1	0	M-1
D+A	0	0	0	0	1	0	D+M
D-A	0	1	0	0	1	1	D-M
A-D	0	0	0	1	1	1	M-D
D&A	0	0	0	0	0	0	D&M
D|A	0	1	0	1	0	1	D|M

d1	d2	d3	dest	Destination
0	0	0	null	No storage
0	0	1	M	RAM[A]
0	1	0	D	D register
0	1	1	MD	RAM[A] and D register
1	0	0	A	A register
1	0	1	AM	A register and RAM[A]
1	1	0	AD	A and D registers
1	1	1	AMD	A and D registers, RAM[A]

j1 (out < 0)	j2 (out $= 0$)	j3 (out > 0)	jump	Effect
0	0	0	null	No jump
0	0	1	JGT	Jump if out > 0
0	1	0	JEQ	Jump if out $= 0$
0	1	1	JGE	Jump if out ≥ 0
1	0	0	JLT	Jump if out < 0
1	0	1	JNE	Jump if out $\neq 0$
1	1	0	JLE	Jump if out ≤ 0
1	1	1	JMP	Jump

Figure 3: Syntax of Hack C-Instructions

C-instructions have a quite complex structure. In contrast to A-instructions, every C-instruction starts with a 1. Then, two bits are unused, and they are also set to 1 by convention. Thus, C-instructions start with three leading 1s.

Afterwards, 7 bits encode the ALU operation, 3 bits encode the destination, and the final 3 bits encode whether a jump takes place.

First, let us focus on the ALU operation. A close look reveals that the last 6 bits are just the ALU control bits, which are embedded directly. E.g., for six 1s, the ALU computes the constant 1, which is what you can find here.

Similarly, for six 0s, the ALU computes the logical And operation.

Recall that our ALU has two inputs, namely x and y. In contrast, Hack instructions may work on the A-register, the D-register, or a memory location as input. As you see in the table here, the seventh bit provides a choice whether to use the A-register or a memory location as second ALU input, while the first input is always wired to the D-register.

Let us now focus on the 3 bits for destinations. Those 3 bits just encode every possible subset of A-register, D-register, and memory location.

Finally, regarding jumps, the 3 bits distinguish the different jump variants seen earlier.

If you want to see examples for binary C-instructions, just load the previous program into the CPU Emulator and switch the view for the ROM from assembly to binary.

5 Conclusions

Let us conclude.

5.1 Summary

- Machine language defines what a machine is able to do
  - Defines abstract interface for computer
    - Hack is 16-bit machine, 15 bits for addresses
    - With A-register, D-register, PC, implicit memory location M
  - Instructions as sequences of 0s and 1s
    - Two type of instruction for Hack: A-instructions and C-instructions
- Assembly language as human readable variant
  - Including symbols for jump targets and variable

Machine language determines the capabilities of a computer system. It establishes an abstract interface for computers, such as the Hack machine, which is a 16-bit computer, where addresses have a size of 15 bits. In addition, Hack features the A-register, D-register, program counter, and the implicit memory location M.

Instructions in machine language are represented as sequences of 0s and 1s, and in the Hack architecture, there are two main types of instructions, namely A-instructions and C-instructions.

Assembly language serves as a human readable version of machine language. It includes symbols for jump targets and variables, making it easier for programmers to write and understand code at a higher level of abstraction.

5.2 Self-Study-Tasks

- Work on the above coding exercise
- Implement "j=j+1" in Hack assembly (see Learnweb quiz)
- Implement `Mult.asm` as part of Project 4 (ignore `Fill.asm`)

Please take a break and program in assembly language. Use the CPU Emulator to verify your assumptions.

Bibliography

Nisan, Noam, and Shimon Schocken. 2005. *The Elements of Computing Systems: Building a Modern Computer from First Principles*. The MIT Press. https://www.nand2tetris.org/.

The bibliography contains references used in this presentation.

License Information

Source files are available on GitLab (check out embedded submodules) under free licenses. Icons of custom controls are by @fontawesome, released under CC BY 4.0.

Except where otherwise noted, the work "Machine and Assembly Language", © 2024 Jens Lechtenbörger, is published under the Creative Commons license CC BY-SA 4.0.

This presentation is distributed as Open Educational Resource under freedom granting license terms.