# *Hack Project*

*(Formerly: The I Computer)*

# *Research project report*

Iftach Amit, School of Computer Science,
The Interdisciplinary Center, Herzlya

## *Abstract*

The study of computer architecture in the past years consisted of several courses such as basic computer architecture (chip design, CISC, RISC etc.), compilers course, and several high-level language design courses (Object oriented, Functional and Logical programming etc.). Those were taught separately with minor interaction between them, or implementation of a whole computer system. This paper describes the Hack project (first called the I Computer), which has been conceived by Prof. Noam Nissan of the IDC and the Hebrew University. The Hack project tries to bind together the wider concept of computer architecture at all levels. The main idea is to design and build a computer, from the ground up, that enables the user to peek into every part of its implementation and experience the redesign and implementation of every component. This is achieved with a special two-part simulator. The first part enables the design of higher-level concepts such as assembly interpreting, memory management, compilers, interpreters and high-level language design. And a second part that enables the design of low-level component from the micro architecture of logical gates, sequential logic and chip design of the basic computer.

# *The Hack computer architecture*

The hack architecture is divided to four parts: the hardware, the machine language level, the virtual machine and the high level language (along with the operating system).

**Hardware**

The hardware of the Hack computer consists of a simple ALU that is capable of performing only two operations - addition and logical AND. The ALU deals with 16-bit numbers represented in two's-complement. This dictates the rest of the system to be based on 16-bit register and memory. The control logic of the ALU enables the manipulation of the inputs (enable, and negate) as well as the output. This scheme practically enhances the capabilities of the ALU to perform all basic operations on its inputs (subtraction, logical OR and NEGATE, getting 1 and 0 for outputs etc.).
The inputs for the ALU are the registers of the computer. The Hack has three registers: A, D and PC. The PC is of course a Program Counter, which is 15-bits long. The D register is a regular 16-bit register, and the A register is similar to D besides the fact that it is used for pointing at the memory for retrieval of data thus incorporates only 15 bits (memory access is in unsigned values). This creates a fourth register namely M[A], which means the value in the memory location which is pointed by A. Each command consists of a 16-bit word whose bits indicate which operation should be performed and where the result is to be sent to (which register).

**Machine language**

The Hack facilitates a low-level assembly language that is basically the binary value of the bit commands as they are sent directly to the control logic of the CPU. Each command is named and thus creates the assembly command set of the Hack (See appendix A for the Hack assembly language with the corresponding decimal value of the commands).

**Virtual Machine**

The Hack virtual machine sits above the assembly language and provides basic programming interfaces for the creation of the high level language. It provides controlled memory access, stack and heap facilities, a new set of "virtual registers" and method context. The Hack compiler uses the virtual machine to create binary program files. These are loaded directly to memory as command sequences at later stages. The virtual machine is stack based as stated earlier and all arithmetic operations are performed on the stack as well as method call handling and parameters. The methods location in memory is recorded into the heap of the corresponding class, and the location of global modules and methods are recorded into a "global table".

**High Level Language**

The high level language of the Hack computer is quite similar in structure to C and incorporates a method structure that reminds Java a bit. The HLL is compiled against the virtual machine and the final

output is usually a loadable binary file. The compilation does not involve any assembly optimizations of any kind and uses only virtual machine commands. A Hack program is consisted of one or more classes, which contain methods. Only a single program can be loaded into memory and run at any given time. The Hack operating system is completely written in the Hack HLL and is accessible for every HLL program that needs the API that it provides. Facilities that are implemented in the OS are arithmetic functions (multiplication and division), array objects, strings, screen and keyboard access, etc.

# *Simulator design and implementation*

**The HACK emulator**

The main purpose of this emulation level is to enable the user to test higher-level concept in an easy manner and on a fast implementation (unlike the chip level which can be used just as well but bears an extensive overhead because of the resolution it enables the user to interfere with).

The emulator has been designed as a set of components that emulate the different hardware components in a high level (i.e. CPU, memory, registers, etc.). These components have interfaces that enable them to interact with each other in a way that conforms to the design of the Hack computer. The components represent the "model" of the computer so that programs could be ran on the in a similar way as to the original design of the Hack architecture.

These components are complemented by another set of components that implement a GUI program. This program enables each model component to have a corresponding GUI component so that the user can interact with them easily. This achieves the second part of the "model-view" design patter and concept in object-oriented programming.

The whole program presents the user with an easy to use interface that enables him to manipulate the computer architecture, view every part of the main hardware component (memory, registers) in a way that exposes the internal design and implementation of the Hack architecture.

This simulation is then used to run a program, written according to the high-level language specification, compiled by the compiler, and loaded as a memory dump in machine language into the hardware memory. Each and every phase detailed in the last sentence, has a specification and an implementation. The user is free to implement every part of the components responsible for every phase according to the specification, thus study a specific part at a time (compiler, virtual machine, and assembly), and test his implementation on the Hack computer, using the program component that he has written.

Because every part is independent of another and the only constraints are the adherence to the specifications and the API of the level "above" and "below" the component in hand, the user is not bound to a certain path of study and is free to decide for himself the best order to learn and confront each phase. This enables a top-down and a bottom-up approach to be used when studying the Hack computer.

**The Compiler**

This level of abstraction implements the compiler, which takes a high-level language program as an input and generates virtual machine level commands as an output.

The compiler is a single-pass compiler. This is due to the high level language specification which took into consideration the fact that the compiler will have to have a simple implementation in order to reduce the complexity of the whole project and concentrate on the overall architecture and give a taste of the compilation process.

The implementation consists of several components:

1. The Lexical Analyzer. This component takes the source code as input and produces tokens (defined as objects) for output. The interface that the lexical analyzer exposes is simple: get the current token, get the next token, and advance one token.
2. The language parser. This component actually performs the hard work of managing tokens and closely interacts with the Lexical Analyzer. The language parser works with one class at a time. This is implemented by passing it the reference to the lexical analyzer that handles the current class. This of course dictates that every class will be written in a file of it's own.

The Language parser also interacts with the virtual machine. The architecture is designed in a way so that the compilation process actually produces bytecode. This is the reason why this interaction is required. In later version of the implementation this should be done in a totally asynchronous mode, where the language parser is to produce a file containing bytecode commands. These files should later be passed to the virtual machine. This will produce a clearer differentiation between the compiler layer and the virtual machine layer.

Implementation of complex look-ahead or handling unknown information (such as going through a loop without knowing where it ends in regard to bytecode) has been implemented using a stack which keeps count of labels in regard to the code structure so that the structure can be compiled into the corresponding bytecode and the nesting will be identical.

All symbols have been handled in a single hierarchy inside an object called "SymbolTable" which implements the handling of scopes and namespaces (three levels).

**The Virtual Machine**

The virtual machine abstraction level is simple in regard to the levels surrounding it (compiler and assembler) because it is actually a mediator between them. Every bytecode command has a corresponding set of commands in the assembly level. Thus the main work of the virtual machine is to take the bytecode input and produce an assembly output.

In the current version this is implemented using a decision tree in an object called "Translator" which takes into account the number of parameters and then finds the appropriate set of assembly commands that correspond to the current bytecode command.

This implementation lacks the flexibility to run as a standalone application for the interaction is completely API based and no wrapper has been built to facilitate this with an external input source (such as a file containing the commands). In later versions this has changed and the VM is actually called with a filename as a parameter, which it needs to parse and extract the bytecode, commands.

# *Chip level design and implementation*

**Design**

The Hack hardware consists of a simple as possible CPU design with as less as possible components to implement. The memory is defined as 16-bit word length base, and there are $2^{15}$ (=32768) words in the Hack memory.

The Hack architecture is implemented using only three registers; a program counter (PC), and two more regular registers (A and D). The first is a 15-bit register, while the other two are 16-bit ones. The PC needs only 15 bits because of the memory length, and the other two registers are full 16-bit registers for memory contents is stored in them. The use of A and D is also defined. A is used to address memory locations, and D is used to retrieve and store memory contents. Therefore is memory access is needed, A is set to the memory address needed to be referenced, and D is used to set/retrieve the value currently stored in the address that A is pointing to.

The main system bus is a 16bit bus connecting the memory with the PC and ALU and is controlled with multiplexers that redirect memory access to/from the appropriate addresses. These multiplexers are controlled by the control logic of the CPU and the PC.

The cycle paradigm of the hack CPU consists of a fetch/execute pair in which first the next command is fetched from memory by using the PC as an address pointer and then the command is executed and may access memory using the A register as an address pointer.

**Implementation**

The hardware has been implemented in the projects first phase by using Java objects to implement the basic hardware components (a NAND gate and a D-Flip-Flop) and objects to represent more complex component which consist of one ore more basic components and complex components. This gave the flexibility to be able to implement very complex circuitry using a recursive approach to the design and implementation of the circuit logic. The first implementation of the hardware as described above did not have a definition language and had to be built by hand from Java files. Later implementations added a "Hardware Definition Language" that eliminated the need to hard-code the circuit design into Java code.

The chip level application simulator is passed a given circuit and starts pumping bits into it; namely simulating a computer clock. This causes the circuit to run and the behavior can then be measured and examined. The later implementation consists of a leaner simulator where the clock itself is controlled from the hardware test language.

This simulator can run the hardware specified before for the Hack computer and can be used to test the design and alter it for further studying the computer architecture. This simulator, on the other hand, has limitations in regard to performance for running the computer in the hardware simulator and expecting it to behave in a fast enough manner to actually interact with is optimistic for the simulator is intended to provide debugging tools and circuit level design and implementation apparatus rather than a platform for running the actual computer.

# *Constraints and solutions*

**Computational model**

The initial computational model (the one which is currently implemented and this document is referring to) has an ALU that is capable of performing shift operations. The shift operation as been used in order to implement multiplication and division operations in the mathematical library offered by the Hack operating system. This has been found to be a tricky but very useful set of commands that enabled the operation of complex operations in a minimal number of CPU cycles.

The operations for division and multiplication for example have the following structure (in high level language – as taken from the actual OS code):

```
while (x > y) {
    let y = y<<;
    let k = k + 1;
}
let y = y>>;
let k = k - 1;

while (k > 0) {
    if (x > y) {
        let buff = buff + 1;
        let x = x - y;
    } else if (x = y) {
        let buff = buff + 1;
        let x = x - y;
    }
    let y = y>>;
    let buff = buff<<;
    let k = k - 1;
}

return (buff>>);
```

This is the division function's core algorithm. It receives two parameters `x`, and `y` to be divided in a manner that the result will indicate x/y (in integer values – without the reminder). `k` is used for counting the size of the remaining number, and `buff` is used for the storage of the actual result.

```
while (x>0) {
        if (1 = x&1) {
                let buff = buff + y;
        }
        let x = x>>;
        let y = y<<;
}
return (buff);
```

This is the multiplication function core algorithm. The computation is very simple as seen in the actual code. X is multiplied by y and again a buffer holds the final computation along the algorithm's operation.

Both functions perform in a magnitude in relation to the inputs *length* in bits rather than the input's *size*.

The recent version of the Hack does not contain a hardware implementation of shifting, thus these operations will have to be implemented in a linear version in the OS. This will present dome new limitations concerning performance and will have to be taken into consideration when implementing the OS.

**Memory size**

This issue has been regarded along the way as a “safe” bet, for 32K words seem enough for this kind of computer and environment.
As it came out, this has been quite a tight assessment for the memory amounts that will be needed for the average general ratio for high-level language code-lines to assembly code-lines exceeded the initial expectations and came up around 1/45. Thus every line of high-level language code compiles to about 45 lines of assembly, where every assembly command incorporates 16-bits (1 word).
Thus 24K of user space memory will accommodate approximately 550 lines of high-level language commands.
This constraint must be taken into proportion, because the allocated 24K of user space RAM is not the only place for the program to run, as the ROM is used to store the OS, and is capable of storing 8K of instructions (approx. 180 HLL code-lines).
Eventually, the memory space proved to be sufficient enough, for the stack and heap that are implemented, do not need large amounts of memory, and several VM optimizations led to smaller code footprint and faster running code, which benefit for the overall project, although performance and optimization were not a part of the issues that we dealt with.

# *Summary*

This project started as a research project with an intention to provide a tool for students studying computer architecture so that the overall design can be understood more clearly and all the computer components can be accessed and redesigned with ease. This was intended to be a simple design in order to enable the user to access all parts without delving too deep into advanced material such as advance compilation or VLSI. The implementation also concentrates on delivering this notion of simplicity and component-independence all the way from the basic parts of the application to the user interface and supporting components.

The project "forced" us to learn extra-curricular material and study deeper aspects of computer architecture in particular and computer science in general, and provided us with a lot of food for thought for several (to say the least) of sleepless nights at the earlier stages of the first coding and designing periods.

Nowadays the project has evolved to be a one-semester course in "Digital Systems Design and Implementation" with a practical approach, in which students experience the design of the Hack computer and implement several parts of it, while having the final implementation as a reference and tool for testing their design.

Still a lot can be done in further extending the design to support networking from the hardware level (implementing a NIC) and maybe experimenting with parallel and distributed systems by extending the component-independent approach further.