IBCM: The Itty Bitty Computing Machine

A one-week module to teach machine language in computing courses

Aaron Bloomfield
University of Virginia
aaron@virginia.edu

William Wulf
University of Virginia
wulf@virginia.edu

ABSTRACT

We present the development and implementation of the Itty Bitty Computing Machine (IBCM), a machine language designed specifically to be taught to lower-level undergraduate students. The presentation of the material takes about one-week of lecture, and allows understanding of all the concepts of machine language without having to deal with the complexity of modern machine language implementations, such as x86 and MIPS. A number of pedagogical aspects are addressed concisely via IBCM, such as treating all data as untyped and performing arithmetic on instructions.

While we are not the first to introduce a short machine language module, we do provide a number of benefits over older versions: a modern browser-based implementation, a full set of pedagogical tools, and a decade of experience teaching this module. All of the necessary materials, including compilers, simulators, and documentation, are available online and licensed through Creative Commons licenses.

Categories and Subject Descriptors

D.3.m [**Software**]: Programming LanguagesMiscellaneous; D.2.6 [**Software Engineering**]: Programming Environments

General Terms

Languages

Keywords

machine language, pedagogy

1. INTRODUCTION

Understanding machine language is vital for any computer science undergraduate. While all undergraduate programs will have their students understand this topic by graduation, the question persists as to how and when best to introduce these concepts to the students. Some institutions have a separate assembly language course that touches on machine language, whereas others include it in a computer architecture course. We are not aware of any institutions that have an entire required computer science course dedicated to machine language. We recognize that hardware-based computer engineering curricula, including ours at the University of Virginia, will have such material as upper-level courses. These courses tend to be specialized computer hardware or computer engineering courses, and are typically not required for a computer science degree.

We present a pedagogical module designed to teach machine language in a single week in a low-level undergraduate course. While the depth of knowledge obtained by the students is limited due to the short time spent on the topic, the students are still able to understand the fundamental concepts. This allows machine language to be introduced to the students early in the curriculum, with a more detailed approach potentially taught in later classes.

We have developed a machine language that we have called the Itty Bitty Computing Machine, or IBCM. This language is intended to be very simple – there are only 16 commands. Yet it is still fully functional, and is a Turing complete programming language.

2. RELATED WORK

We are certainly not the first to propose a simplified machine language as a pedagogical tool. Andrew Tanenbaum's original 1984 text, *Structured Computer Organization* – now in its 5th edition – presents the Mic-1 micro architecture and the Mac-1 machine language [10]. Indeed, the Mac-1 machine language has many similarities with IBCM – this is not surprising, as there are many common elements that must be present in all small instruction set machine languages. Further research in that decade presented implementations of those languages [5, 6]. The Mic-1 simulator, which is focused on assembly language, is available online [9]. Simulators for the Mac-1 machine language do exist, but seem to be independently developed, and without a modern set of implementation software.

A number of high quality simulators exist at the assembly level. Pep/8 [11, 12] is a 16-bit CISC architecture designed to teach assembly language concepts. While it can be used for machine language – and can trace program execution at the machine language level – the primary pedagogical design is at the assembly language level. Another example is SPIM, which is a full featured MIPS 32-bit simulator [4].

Additional research on machine language simulators has

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGCSE'11, March 9–12, 2011, Dallas, Texas, USA.
Copyright 2011 ACM 978-1-4503-0500-6/11/03 ...$10.00.

often focused on the register transfer level [7], or is restricted to a single client operating system [1].

More recent research by Stone has focused on a similar machine language implementation [8]. Although developed independently, our research can be seen as an extension of Stone's, as we add a number of additional aspects: significant pedagogical tools, a fully downloadable package so this system can be used in any course, a proof of Turing completeness, and a discussion of pedagogical concerns.

We are not aware of any machine language simulators that are available with the set of modern tools that we present with IBCM.

3. THE IBCM LANGUAGE

The Itty Bitty Computing Machine is an accumulator-based machine language with 16-bit two's complement integer arithmetic. There are no registers other than the accumulator. IBCM has 4096 words of memory, and each word is also 16 bits.

All instructions are likewise 16 bits: the first four bits are the opcode, and the rest are interpreted differently, depending on the opcode. The instruction format can be seen in Figure 2.

0	1	2	3	4 5	...	15	
0	0	0	0	(unused)			halt
0	0	0	1	I/O op	(unused)		I/O
0	0	1	0	shift op	(unused)	count	shifts
opcode				address			others

Figure 2: IBCM instruction format

An opcode of 0x0 halts the IBCM. The remaining 12 bits are not used by this instruction, and can hold any value.

An opcode of 0x1 will perform either an input or output operation. All input is written to the accumulator, and all output is read from the accumulator. The two bits following the opcode specify which operation: the 5th bit specifies either a read (if 0) or a write (if 1), and the 6th bit specifies if the value being read or written is a four digit hexadecimal word (if 0) or an ASCII character (if 1).

An opcode of 0x2 will perform a shift operation on the bits in the accumulator. The bit following the opcode specifies whether the operation is a shift (if 0) or a rotate (if 1). A rotate takes the bits that "fall" off one "end" of the 16-bit word, and move them to the empty spots on the other "end" of the word. The 6th bit of the instruction specifies if the shift or rotate operation is to the left (if 0) or right (if 1). The last 4 bits of the 16-bit word (called 'count' in Figure 2) specify how many bit positions the shift or rotation should move.

The remaining 13 opcodes (0x3 to 0xf) are classified as 'address' instructions, as the remaining 12 bits generally specify the memory address that the opcode will use. A few of the instructions do not need the 12 bit address: in particular, the `not` and `nop` operations. All of 13 these 'address' instruction opcodes are shown in Figure 1. In that figure, note that `a` represents the accumulator register, and `mem[]` represents memory. Of these 13 operations, the first 8 manipulate data, and last 4 perform program flow control; the `nop` instruction does nothing. Note that `PC`, the program counter, is the address of the current instruction being executed.

An IBCM file is simply a text file, and the simulator only reads in the first 4 bytes of each line; the remaining characters on the line are ignored. In the sample program that is shown below (in Figure 4), additional opcodes are used for clarity. These are so that one can read an assembly-like opcode for each line.

- `dw` (for "declare word"), for declaring variables
- `readH` and `printH` are for reading or writing a hexadecimal value
- `readC` and `printC` are for reading or writing an ASCII character
- `shiftL` and `shiftR` are for the shifts
- `rotL` and `rotR` are for the rotations

All the functionality of a modern procedural programming language can be implemented using IBCM. Conditionals (if-then-else) are implemented through comparing a number!– or a number minus a constant!– to zero, and using the various jump commands (`jmp`, `jmpe`, `jmpl`) to access the different parts of the conditional. As in any assembly or machine language, iteration is achieved through looping, such as the `jmp` command. Subroutines can be implemented through the `brl` instruction, which stores the return address in the accumulator upon jumping to the subroutine start address. The `brl` command is discussed in more detail below, as well as an IBCM example that uses subroutines.

A specific design decision with IBCM was to include only basic operations – for example, multiplication and division are not included, but can be replicated by using repeated addition or subtraction. We wanted students to see that any program, no matter how complicated, can be broken down into very simple instructions. Indeed, this is the point of the concept of Turing machines, but they are often not taught when students are first seeing machine language.

4. EXAMPLE PROGRAMS

We present one example IBCM program in its entirety here in Figures 3 and 4. We discuss two other programs in this article, and their full program listings can be found online [2].

The first program will compute the sum of the integers 1 through n, where n is read from the keyboard; the resulting sum is printed to the screen. The program then halts after printing the sum. The C++ code for the program is shown in Figure 3 – this is presented to help show the conversion to IBCM.

```
int main(void) {
  int n, s = 0;
  cin >> n;
  for ( int i = 0; i <= n; i++ )
    s += i;
  cout << s << endl;
}
```

Figure 3: C++ summation program

The IBCM program is shown in Figure 4. Note that the only part of the program that the simulator reads in is the

Hex	Opcode	HLL-like meaning	English explanation
3	load	a:= mem[addr]	load accumulator from memory
4	store	mem[addr] := a	store accumulator into memory
5	add	a := a + mem[addr]	add memory to accumulator
6	sub	a := a - mem[addr]	subtract memory from accumulator
7	and	a := a & mem[addr]	bitwise 'and' of memory and accumulator
8	or	a := a | mem[addr]	bitwise 'or' of memory and accumulator
9	xor	a := a ⊕ mem[addr]	bitwise 'xor' of memory and accumulator
A	not	a := ∼a	bitwise complement of accumulator
B	nop		do nothing (no operation)
C	jmp	goto addr	jump to 'addr'
D	jmpe	if a = 0 goto addr	jump to 'addr' if accumulator equals zero
E	jmpl	if a < 0 goto addr	jump to 'addr' if accumulator less than zero
F	brl	a := PC; goto addr	set accumulator to PC; jump (branch) to 'addr';

Figure 1: IBCM instructions

first four characters on the line. The rest of the line in the text file is solely for comments. The column headers shown in the figure are heavily abbreviated to fit in the width of a column, but are, in order: the actual 4 hexadecimal digit memory value, the hexadecimal location (used for determining jump targets and variable addresses), the label (used to refer to jump and variable targets), the opcode (from Figure 1), the target address (which refers to a given label), and any English comments.

Mem	Loc'n	Label	Opcode	Adr	Comments
C006	00		jmp	init	jmp past vars
0000	01	i	dw		int i
0000	02	s	dw		int s
0000	03	n	dw		int n
0001	04	one	dw		
0000	05	zero	dw		
1000	06	init	readH		read n
4003	07		store	n	
3004	08		load	one	i = 1
4001	09		store	i	
3005	0A		load	zero	s = 0
4002	0B		store	s	
3003	0C	loop	load	n	if i>n, jmp xit
6001	0D		sub	i	
E016	0E		jmpl	xit	
3002	0F		load	s	s += i
5001	10		add	i	
4002	11		store	s	
3001	12		load	i	i += 1
5004	13		add	one	
4001	14		store	i	
C00C	15		jmp	loop	goto loop
3002	16	xit	load	s	print s
1800	17		printH		
0000	18		halt		halt

Figure 4: IBCM summation program

Another program that was developed computes the product of two numbers, x and y, through a recursive multiplication subroutine that uses only addition. Both x and y are read from the keyboard, and the resulting product is printed to the screen. The program is just over 100 lines of IBCM opcodes, and is available online [2].

The C++ code for the multiplication program can be seen in Figure 5. We did not create a tail recursive `multiply()` routine, as the IBCM compiler and language is (intentionally) far too simple to optimize for tail recursion.

```
int mult(int x,int y) {
  if ( y == 0 )
    return 0;
  else
    return x+mult(x,y-1);
}
```

```
int main(void) {
  int x, y;
  cin >> x;
  cin >> y;
  cout << mult(y,x)
       << endl;
}
```

Figure 5: C++ multiplication program

This IBCM program creates a stack similar to x86: the stack starts at the end of addressable memory, and grows downward. An activation record is created for each recursive call, which consists of the two parameters and the return address – other fields typically in an activation record (e.g., backup of registers) are not necessary in IBCM. The `brl` instruction was used to allow for subroutine calls – the return address is saved in the accumulator, and is stored on the stack immediately upon subroutine activation. We were able to run the program with the second parameter (which is decremented in each recursive call) set as high as 1,243 (0x4db), beyond which point IBCM runs out of available memory for the stack.

This recursive multiplication program is beyond what we would expect of a student to be able to program after a one-week introduction to IBCM. However, it is very illustrative of two important points about IBCM. One is that complex functionalities (such as multiplication) can be achieved by using only the simple capabilities available in IBCM (such as addition). The other is that subroutines are fully realizable in IBCM.

5. TURING COMPLETENESS

We were conflicted as to how much to discuss about Turing completeness in this article. IBCM is similar to a Random Access Stored Program (RASP) machine [14], which is itself Turing complete, so it is perhaps not surprising that IBCM is also Turing complete. However, we felt that breaking down a complex task (a Turing machine simulator) and programming it into IBCM was a worthwhile task to discuss,

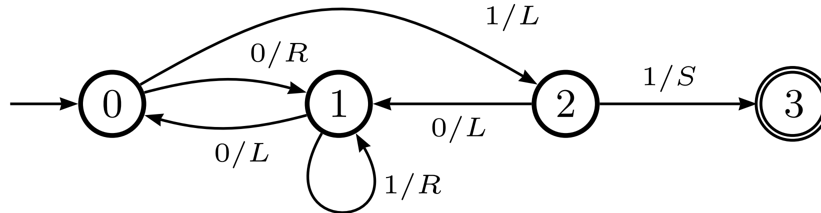

Figure 6: Four state Busy Beaver automaton

as it emphasizes a primary design goal of IBCM – that you can take any program and write it in IBCM.

Obviously, no physical computer with a finite amount of memory can be truly Turing complete. Thus, we will instead show that the IBCM computational model is Turing complete.

We define the *IBCM computational model* as the same IBCM computer defined above, but allowing any sized integer to be held in a single memory location, as well as having an infinite amount of memory. Thus, other fields that make up part of a given instruction, such as the 'address' or 'count' fields (see Figure 2), can also hold any size integer.

Given this model of computation, we will show how to simulate a Turing machine in IBCM. Hopcroft and Ullman [3] define a Turing machine as a 7-tuple $M = \langle Q, \Gamma, b, \Sigma, \delta, q_0, F \rangle$ where:

- Q is a finite set of states
- Γ is the finite set of allowable tape symbols
- $b \in \Gamma$ is the blank symbol
- $\Sigma \subseteq \Gamma \setminus b$ is the set of input symbols
- $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$ is the transition function
- $q_0 \in Q$ is the initial state
- $F \subseteq Q$ is the set of final states

We will make two modifications to the above definition, to allow for ease of implementation in IBCM. We will allow a no-shift transition of S (in addition to L and R), which will not move the tape. The Turing machine programs allowed by the no-shift are equivalent to the ones described above [15]. Furthermore, we will define f, a single final state, which all states in F move to on a no-shift transition.

To represent the transition functions in IBCM, we will represent state $q \in Q$ and symbol $\Sigma \in \Gamma$ each as a (16-bit) word in IBCM. Thus, states and symbols will be a single integer each. While this limits the number of states (and symbols) to $2^{16} = 65,536$ in our IBCM implementation, it is not limited in the formal IBCM computational model. Thus, one can encode any amount of states, symbols, and transition functions into IBCM's memory.

To simulate a Turing machine, we will define an arbitrary memory location to represent the current state of the Turing machine, and another arbitrary memory location to contain the current address of the head of the tape. The transition function quintuples, δ, will start at a specific (but arbitrary) memory address, take up five words each, and will contain the five parts listed above $(Q, \Gamma, Q, \Gamma, \{L, R, S\})$. The tape itself will start at different arbitrary memory location. Furthermore, we define a initial state q_0 and a (single) final state f. The blank symbol b will be an arbitrary value, such as -1 (0xffff in 16-bit 2's complement integer).

Any Turing machine that requires a significant amount of tape will need to be a one-way tape Turing machine, as the program code and state transitions will lie at a lower address than the initial head position. Thus, only a finite amount of tape space is available in the lower memory address direction. The particular automaton example that we provide, below, uses a two-way tape, but that is because we know the finite amount of tape space necessary. Note that one-way tape Turing machines are equivalent in computational power to two-way tape Turing machines [3].

What is needed, then, is an IBCM program that will iterate through the following steps:

- Read the current state s, initially set to the start state. If the current state is the (single) final state, then exit.
- Read the current head position.
- Read the current input symbol t at the head position
- Search the list of transition functions until the appropriate one is found, based on the current state s and the input symbol t.
- Perform the action specified in the transition function by updating the state s, writing the specified symbol to the tape position, and then moving the tape left or right (or, on an S, not at all).

We have developed such a program, which we describe here. The full listing of the program is available online [2]. The program consists of 67 IBCM commands, and 15 variables – note that numeric constants are considered variables in an IBCM program. This program only used half of the instructions: `halt`, `load`, `store`, `add`, `sub`, `nop`, `jmp`, and `jmpe`.

To test the Turing machine, we choose a four state Busy Beaver automaton, which is described in more detail in the Wikipedia page on Turing machine examples [16]. The Mealy machine finite automaton is shown in Figure 6. For each transition, the input symbol (0 or 1) is shown, along with the tape direction to move (L, R, or S). Note that in this automaton, upon each transition a 1 is written to the output tape; this is not shown in the figure to improve clarity. Also recall that the S transition means to not move the tape, and is used only on the transition to the final state.

Our implementation can utilize a two-way tape, although the tape in one direction is finite. We define memory address 1 as the current state variable, and address 2 to store the location of the current head position. The transitions start at address 0x060, as our program takes up 82 (or 0x052) instructions. The states are numbered as per the diagram in Figure 6, with 0 being the initial state and 3 being the final state. The program has constants that specify both the initial and final states of the automaton.

The encoding of the automaton shown in Figure 6 is very straightforward. The transition from 2 to 1, executed on an input symbol of 0, will print 1 to the tape and then move the tape to the left. The quintuple to be encoded is $Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R, S\}$. The respective values for this transition are $2, 0, 1, 1, 0$; we map the integer values $\{0, 1, 2\}$ to, respectively, the transition directions $\{L, R, S\}$.

6. PEDAGOGY

IBCM was originally developed to complement our CS 3 course entitled *Program and Data Representation*, which is still taught today. This course shows how one represents both data and program code from high levels – such as abstract data types – all the way down to the lowest (software) level, which is the IBCM machine language.

IBCM allows students to easily make the mental connection between assembly opcodes and the machine language that they get translated into. At the University of Virginia, we follow the presentation of IBCM with a two-week introduction to assembly language. This allows students to understand both machine language, as well as a modern processor's assembly language (we use Intel x86), without having to delve into the details of x86 machine language.

Students are exposed to a number of concepts during the IBCM module that they often have not seen previously. They become aware that in both assembly language and machine language, data is untyped, and the operations on the data determine the type – this is quite different than the typed high-level programming languages to which they are accustomed. By this point in our course, students have been exposed to how a 32-bit value can be interpreted either as a two's-complement signed integer or an IEEE 754 floating point number.

Another concept taught through IBCM is self-modifying programs. A non-trivial IBCM program requires arithmetic on instructions – in fact, only the first example program shown in Figure 4 did not use this feature. Array indexing, for example, requires starting with a `load` instruction, and adding to that value both the base address of the array and the current index. This value is then stored in memory which is shortly thereafter executed. The second example program provided to the students uses this feature. While many systems explicitly try to prevent self-modifying code, as that is an exploit used by a significant amount of malware, it is still a concept that the students should be familiar with.

The development of self-modifying IBCM programs leads to another pedagogical goal of the course: the interplay between data and program code. Indeed, there is little difference between data and program code, other than the values (obviously), and how it is interpreted (and, in modern systems, what segment of an executable in which the data is found). This concept seems trivial to instructors, but is one that students who have only programmed using high-level languages are often unfamiliar with.

What IBCM does not teach, of course, is how binary machine language instructions are executed on the processor. Understanding of this material is typically beyond all but senior-level undergraduate courses; at many institutions, this is also outside the standard computing curriculum.

7. RESULTS

At the University of Virginia, this module is taught about half-way through our CS3 course, which is where we teach data structures. Our CS1 and CS2 course are both in Java. At this point in the CS3 course, the students have learned to implement a number of data structures in C++. We introduce machine language using IBCM for one week, follow that with two weeks of assembly programming (x86), and then return to C++ for the remainder of the semester. The lectures used to teach IBCM typically take three 50-minute class periods. This is followed by an IBCM lab the following week. All of the lecture slides and labs are available online [2].

We have taught machine language using IBCM in our CS3 class for over a decade. Student reactions to IBCM have varied greatly. With the usage of the modern tool set presented in this article, those reactions have generally been positive, as shown below While they often do not like being constrained to such a limited set of instructions, they see the purpose of learning machine language, and generally enjoy the IBCM assignments. We have found that the quality of the software tools is directly related to student perception of IBCM – in the past, when our IBCM simulator was less refined, student reaction was significantly more negative.

Objective comparison with other programming languages, including assembly language, are difficult due to the vast differences between both the capabilities and the required learning curve. We have instead focused on subjective assessment. In the most recent semester in which IBCM was used (fall of 2010), six questions were asked of the students. All questions were asked on a Likert scale, where 1 means strongly agree, 2 is agree, 3 is neutral, 4 is disagree, and 5 is strongly disagree. For all the questions, $n = 89$.

The first two questions focused on how much the students felt they learned from the IBCM module. The middle two questions focused on the ease of use and enjoyability. And the last two questions focused on how worthwhile this module was, both for this course and future courses.

Question	**Avg**	**Stdev**
IBCM increased my understanding of the basics of machine language	1.67	0.58
IBCM increased my understanding of how computers work at a low level	1.89	0.65
IBCM was easy to use, once I got the hang of programming in it	1.92	0.88
I enjoyed learning IBCM	2.01	0.98
Considering what was taught, IBCM was a worthwhile module to have in this course	1.75	0.68
IBCM should be used in future iterations of this course	1.76	0.72

The results clearly show that the module was generally well received. The lowest score, enjoyment of learning IBCM, still was in the 'agree' category.

8. MATERIALS AVAILABLE

We have developed and made available a wide range of materials for the purpose of teaching the IBCM module. The materials are all released under various Creative Commons licenses. They are available online [2], and consist of:

- A PowerPoint slide set to introduce the concepts during a lecture-based course. This 42 slide set takes about three 50-minute lectures to present.
- A *Principles of Operation* document, which describes the IBCM computer and language, and how to write a program. It covers similar content to the lecture slides.
- Sample programs, one of which was shown in Figure 4, above. We also provide sample programs on array indexing, for example. All the programs mentioned in this article are available online.

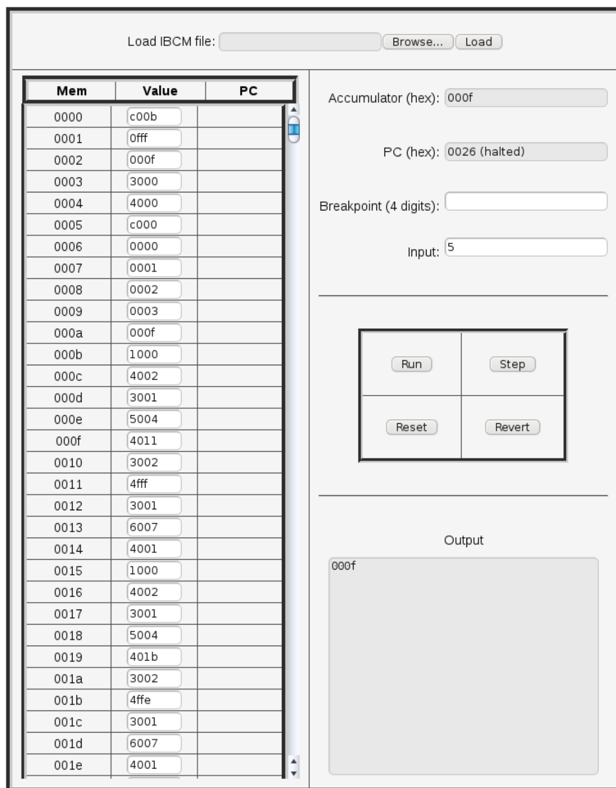

Figure 7: Web-based IBCM interface

- Sample student assignments, which require students to write additional IBCM programs beyond the sample programs provided. One of the assignments is to write a quine, or an IBCM program that will print itself out – the smallest quine produced is nine IBCM commands.
- An online PHP/Javascript simulator, which is the primary way that the students program in IBCM. This is shown above in Figure 7. The PHP is used to allow loading of an IBCM program from a text file; the Javascript implements the IBCM simulator in the browser itself. The simulator works across all major browsers on all major operating systems.
- A C++-based command-line program, which can both compile and execute IBCM programs. Not surprisingly, this is much faster than the online simulator. This is particularly useful for automated compilation and execution of IBCM programs for grading, or for very long programs, such as the recursive multiply routine described above.

Furthermore, a GUI-based tool for executing IBCM programs is available separately [13]. This tool allows for drag-and-drop loading of IBCM files into the GUI, and compiles natively for each operating system.

We very specifically have not developed an IBCM assembler, which would take in the opcodes in an assembly language format and output hexadecimal machine code. The purpose of IBCM is to teach the students machine language; given an IBCM assembler, this module ends up being just a different assembly language for the students to learn.

9. CONCLUSION

We have used IBCM for over a decade at the University of Virginia in our CS3 class. During that time, we have refined it into the pedagogical tool presented here. With the set of current pedagogical tools provided, we have found it to be an effective means of teaching the basic concepts of machine language without having to go into the complexity of modern machine languages that is beyond the scope of a lower-level undergraduate course. All of the necessary materials are available online for adoption at other institutions.

10. REFERENCES

[1] B. Lewis Barnett, III. A visual simulator for a simple machine and assembly language. In *SIGCSE '95: Proceedings of the twenty-sixth SIGCSE technical symposium on Computer science education*, pages 233–237, New York, NY, USA, 1995. ACM.

[2] Aaron Bloomfield. Itty bitty computing machine online. `http://www.cs.virginia.edu/~asb/ibcm/`.

[3] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Publishing Company, 3rd edition, 2006.

[4] James Larus. SPIM: A MIPS32 Simulator. `http://pages.cs.wisc.edu/~larus/spim.html`.

[5] Jerry E. Sayers and David E. Martin. A hypothetical computer to simulate microprogramming and conventional machine language. *SIGCSE Bull.*, 20(4):43–49, 1988.

[6] Delmar E. Searls. An integrated hardware simulator. *SIGCSE Bull.*, 25(2):24–28, 1993.

[7] Dale Skrien and John Hosack. A multilevel simulator at the register transfer level for use in an introductory machine organization class. *SIGCSE Bull.*, 23(1):347–351, 1991.

[8] Jeffrey A. Stone. Using a machine language simulator to teach cs1 concepts. *SIGCSE Bull.*, 38(4):43–45, 2006.

[9] Andrew Tanenbaum. Mic-1 download website. `http://www.ontko.com/mic1/`.

[10] Andrew S. Tanenbaum. *Structured Computer Organization*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1984.

[11] J. Stanley Warford and Chris Dimpfl. The pep/8 memory tracer: visualizing activation records on the run-time stack. In *Proceedings of the 41st ACM technical symposium on Computer science education*, SIGCSE '10, pages 371–375, New York, NY, USA, 2010. ACM.

[12] J.S. Warford. *Computer Systems*. Jones & Bartlett Learning, 2009.

[13] Jacob Welsh. JWelsh's useful programs. `http://www.eemta.org/~jwelsh/progs/`.

[14] Wikipedia. Random access stored program machine. `http://en.wikipedia.org/wiki/Random_access_stored_program_machine`.

[15] Wikipedia. Turing machine. `http://en.wikipedia.org/wiki/Turing_machine`.

[16] Wikipedia. Turing machine examples. `http://en.wikipedia.org/wiki/Turing_machine_examples`.