6-2007

A Simple and Affordable TTL Processor for the Classroom

David Feinberg
The Harker School

A Simple and Affordable TTL Processor for the Classroom

Dave Feinberg
The Harker School
500 Saratoga Ave
San Jose, CA 95129
(408) 249-2510
davef@harker.org

ABSTRACT
This paper presents a simple 4-bit computer processor design that may be built using TTL chips for less than $65. In addition to describing the processor itself in detail, we discuss our experience using the lab kit and its associated machine instruction set to teach computer architecture to high school students.

Categories and Subject Descriptors
C.1.m [**Processor Architectures**]: Miscellaneous

General Terms
Design, Languages

Keywords
Architecture, Processor, Education

1. INTRODUCTION
In the fall semester of 2005, eighteen high school students at The Harker School successfully connected TTL chips on solderless breadboards to build their own computer processors. Most of the students had no practical electronics experience, beyond a basic understanding of serial and parallel circuits. We created this computer architecture course to complement the high-level software focus of the AP computer science course these students had already completed. Our primary goals for the course were:

- to explore what a computer is
- to examine the interplay between hardware and software
- to link machine code to high-level program constructs

As an undergraduate, the author was among the last of MIT's 6.004 Computation Structures students to use the MAYBE—a lab kit for building an 8-bit TTL computer processor. Future 6.004 students would use hardware simulation software instead. Although such simulation tools present a powerful and affordable way to study computer hardware, we believe there is greater value in working with physical hardware, which provides the most convincing means for students to internalize the subtle interplay between software and hardware.

2. LAB KIT REQUIREMENTS
In setting out to find a hardware lab kit, we identified three key requirements. The kit must (1) be understandable, (2) require minimal assembly time, and (3) be purchased at minimal cost. Striving to keep the processor as simple as possible naturally lead to meeting all three requirements. To achieve a simple design, however, it was necessary to give up many features of more serious processors. As long as our processor's instruction set comprised a universal programming language, we were willing to sacrifice the ability to run large or even useful programs. Optimizations such as caching and pipelining would only obscure the core computer science concepts we wished to illuminate.

Like MIT's MAYBE, nearly all university hardware lab kits and numerous amateur-designed processors feature the TTL chip set. Although TTL has stringent voltage requirements, its ability to withstand static charge made it ideal for our course. Because TTL chips typically come in 4-bit packages, MIT's 8-bit MAYBE processor required 2 ALU chips, 2 counter chips, etc. Students assembling MAYBE kits frequently found themselves working through the night to complete their wiring. As this level of time commitment is unacceptable for a high school class, we settled on a 4-bit datapath for our processor. We feel certain that students can learn as much from building a 4-bit processor as an 8-bit one, and a 4-bit processor means half as much time spent wiring and debugging, half as much money spent on chips, and a solderless breadboard of half the size and cost.

A search through simple processor designs used in various university computer architecture courses and those published online by amateurs revealed only a handful of 4-bit processors. As each of these was either too complex or insufficiently documented, we set out to design our own processor. In the course, we referred to it affectionately as "the CHUMP" ("Cheap Homebrew Understandable Minimal Processor").

3. DATAPATH
In all aspects of design, we aimed to identify the simplest solution. Designing an understandable processor meant using a RISC architecture, in which a simple datapath and small instruction set could give rise to complex behavior. We decided that the simplest design would involve fetching and executing the instruction in the same clock tick. Since both the instruction and

the data it manipulated in RAM would need to be accessed in a single tick, we stored our program separately in an EEPROM.

Each CHUMP instruction consists of two parts: an op-code (indicating which operation the processor should perform) and a 4-bit constant/immediate value (used as a data value, RAM address, or program line number). Hence, the CHUMP can only manipulate 4-bit data values, representing numbers in the narrow range from 0 to 15 (or -8 to 7). Likewise, it can only access 16 locations in RAM. Finally, the use of a single 4-bit program counter means that programs cannot exceed 16 lines.

Although we originally sought to limit the instruction set to just four or five essential instructions, we found that supporting a set of 14 instructions did not complicate our design. This larger instruction set allows students to write programs that would be both readable and compact. Thus, a 4-bit op-code is required to distinguish among the 14 instruction types, and each CHUMP instruction uses 8 bits, as shown in Figure 1.

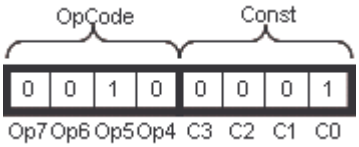

Figure 1: Anatomy of a CHUMP Instruction

The CHUMP processor design is illustrated in Figure 2, with each of the major components corresponding to a single TTL chip. All bold arrows indicate 4-bit connections (4 wires). The program counter chip (labeled "PC") stores the number of the instruction currently being accessed from the Program ROM. A multiplexer selects between the instruction's constant value and the data value read from the RAM chip. The selected value may then be used as:

- the second operand to the ALU, which may increase, decrease, or replace the value stored in the accumulator register (labeled "Accum").
- the next instruction number
- the next address to read/write (stored temporarily in flip-flops labeled "Addr")

When a value is read from memory, it is available for use in the next clock tick by the subsequent instruction. Thus, it takes two clock ticks and two different instructions to first read from an address in RAM and then load this value into the accumulator. However, only a single instruction is required to store a value in the accumulator to memory. Not surprisingly, this asymmetrical behavior proved to be a stumbling block for students.

The overall datapath was selected both for its simplicity and its computational power. It allows the program execution to jump to a line number stored in RAM, and to use a value stored in RAM as the next RAM address to access. It also allows the value in the accumulator to be used as the next line number or memory address, provided the programmer first stores it to RAM.

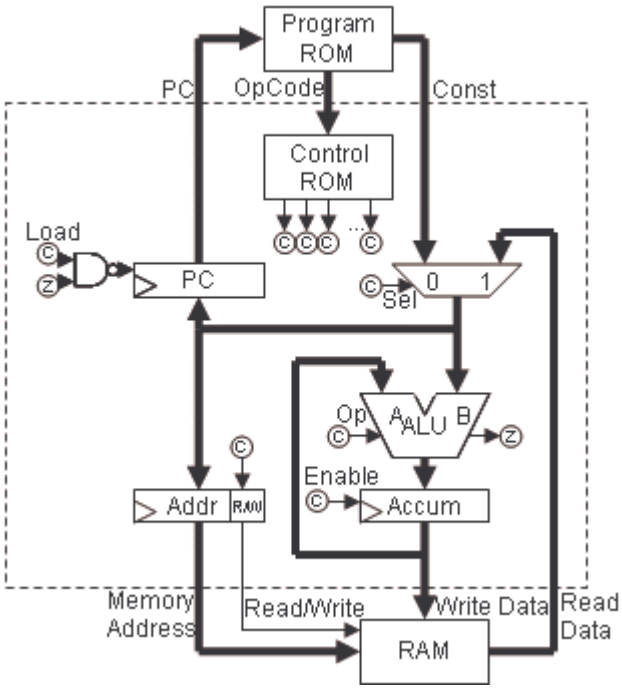

Figure 2: The CHUMP Processor

This processor design suffers from several major limitations, the most significant being the maximum program length of 16 instructions (which cannot even support the simplest meaningful program). Equally frustrating is the inability to write interactive programs, which might pause for user input from switches. Finally, the CHUMP instruction set does not support comparison operations, which could be used to perform arithmetic on larger numbers.

Table 1. CHUMP Constant Instructions

Instruction	Summary	Description
LOAD 0000	`accum = const; pc++;`	load constant into accumulator
ADD 0010	`accum += const; pc++;`	add constant to accumulator
SUBTRACT 0100	`accum -= const; pc++;`	subtract constant from accumulator
STORETO 0110	`mem[const] = accum; pc++;`	store accumulator value to constant address
READ 1000	`addr = const; pc++;`	read from constant address
GOTO 1010	`pc = const;`	jump to constant instruction address
IFZERO 1100	`if (accum == 0) pc = const; else pc++;`	jump to constant instruction if accumulator is zero

Each of these limitations is easily addressed by the addition of a couple more chips. Nonetheless, we elected to keep the core processor as simple as possible, and then to allow students to make such modifications at the end of the course. Although several students were capable of implementing such enhancements, ultimately none chose to do so (a good indication that the limited CHUMP design was sufficient for our course).

4. INSTRUCTION SET

The CHUMP instruction set features seven key operations, each of which comes in two flavors: constant and memory. For example, there is an `ADD` command for adding a constant to the accumulator, and another `ADD` for adding a value from memory to the accumulator. The 4-bit constant portion of the instruction is ignored by the seven memory commands. Table 1 describes the seven constant commands. The corresponding memory commands operate similarly on a memory value, and have a `1` in the op-code's low-order bit.

For example, the following program increments the value in RAM location 2 repeatedly. Used properly, every `READ` command should be followed by a memory command, and every memory command should be preceded by a `READ` command.

```
0:  10000010    READ 2
1:  00010000    LOAD IT
2:  00100001    ADD 1
3:  01100010    STORETO 2
4:  10100000    GOTO 0
```

5. CONTROL LOGIC

We now describe the portion of the processor that examines the 4-bit op-code and uses it to control the operation of each chip. The CHUMP has 5 control points:

- Multiplexer: may select the constant or memory value
- ALU: may perform one of several arithmetic operations
- Accumulator: may load or ignore the ALU's output
- RAM: may perform either a read or write operation
- Program Counter: may load a new value or increment

Because the ALU must perform multiple operations, it requires several control bits (6, in the case of the 74LS181 ALU chip we used). Note also that the control bit for the RAM must first pass through a flip-flop, so that it is clocked together with the 4 address bits. (We used a single chip for all 5 flip-flops.)

We used a single bit to control jumps. This jump bit indicates either (a) the program counter should increment, regardless of the value of ALU output pin *Z* (which can be used to determine if the accumulator value is zero), or (b) the program counter's behavior should depend on *Z*. (Consequently, an extra logic gate is needed to determine the program counter's *load* input from the jump and *Z* bits.) Table 2 summarizes the values of the control bits used by the CHUMP. The multiplexer's control bit (not shown) is simply the low order op-code bit.

Note that the ALU function for a `GOTO` command should always cause *Z* to indicate a zero value, while the function for an `IFZERO` command should cause *Z* to reflect whether ALU input *A* is zero. (For the 74LS181 chip, the relevant *A=B* pin indicates whether the ALU's output is `1111`. Therefore, the ALU should perform the *Logic 1* operation for the `GOTO` instruction, and *Not A* for the `IFZERO` instruction.)

Table 2. Control Logic

Instruction	ALU	Accumulator	RAM	Jump
`LOAD`	*B*	load	read	next
`ADD`	*A* plus *B*	load	read	next
`SUBTRACT`	*A* minus *B*	load	read	next
`STORETO`		ignore	write	next
`READ`		ignore	read	next
`GOTO`	[see text]	ignore	read	load if *Z*
`IFZERO`	[see text]	ignore	read	load if *Z*

Finally, the 4 op-code bits output by the program ROM must be fed through a layer of combinational logic so as to determine the values of the many control bits indicated above. Students chose to use an additional ROM to perform this task.

6. LAB KIT

A fully assembled CHUMP processor is shown in Figure 3.

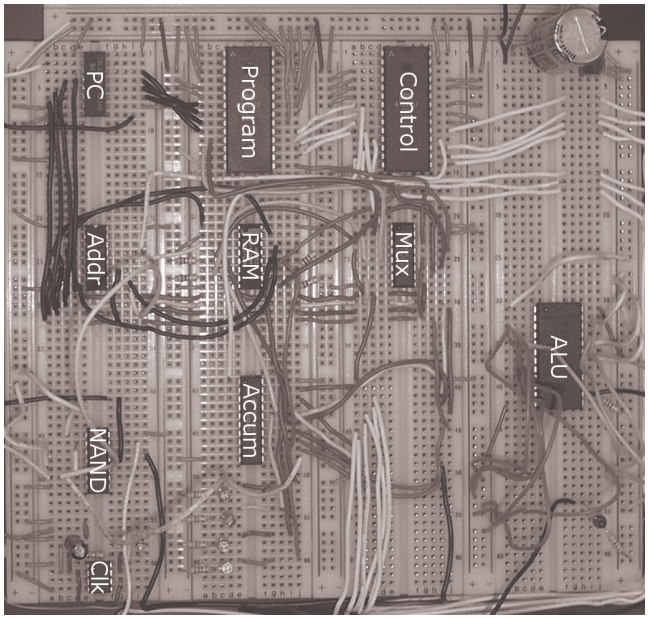

Figure 3: A high school student built this CHUMP lab kit.

The clock circuit we used ran at just 1 Hz, allowing students to follow the execution of their programs in real-time. In practice, we rarely used the clock circuit, relying instead on a simple RS-circuit to serve as a manual clock source. Only at the end of the course, when the entire kit had been fully tested, did students

replace the manual clock with the real one to watch their computer run automatically.

Our piecemeal lab kit featured the IC chips listed in Table 3. Most are easily replaced by other available TTL-compatible chips.

Table 3. Integrated Circuits Used

Chip #	Description	Usage
555	Timer	clock source
74LS00	Quad 2-Input NAND	jump bit logic
74LS157	Quad 2/1 Data Selector	select constant/memory
74LS161	4-Bit Counter	program counter
74LS174	Hex D Flip-Flop	next read/write address
74LS181	Arithmetic Logic Unit	add/subtract, test if zero
74LS377	Octal D Register	accumulator register
74S289	64-Bit RAM	data storage
AT28C17	2k × 8 Parallel EEPROM	program, control logic

Keeping lab costs low was essential to make digital electronics accessible to a high school classroom. As students would be required to pay for their lab kits, we were determined to keep the cost of a lab kit comparable to that of a text book. While most such university lab kits cost many hundreds of dollars, our minimal kit wound up costing less than $65 each. Table 4 shows a breakdown of the major costs involved.

Table 4. Major Lab Material Costs

Item	Cost	Recommended Vendor
solderless breadboard	$20	CircuitSpecialists.com
four 9-volt batteries	$6	Target, Walgreens
22- or 24-gauge solid wire	$5	Fry's Electronics
two AT28C17 EEPROMs	$4	Jameco Electronics
wire stripper	$4	Jameco Electronics
large plastic bin	$2	Wal-Mart
needle-nose pliers	$2	Orchard Supply Hardware
74LS181 ALU	$2	Jameco Electronics

Although the CHUMP can be assembled on a much smaller board, a solderless breadboard with 3000+ contact points was selected to give students greater flexibility in laying out components. Such boards typically sell for $35, so finding them online for $20 was critical in keeping the lab kit affordable.

A 7805 voltage regulator (and requisite capacitors) provided a steady 5-volt power source for the various chips. It requires an input of 7.5 – 12.5 volts to work effectively. Rather that using an AC/DC adapter (easily destroyed by a temporary short), we opted to use a single 9-volt battery, and purchased several per lab kit.

The remainder of the lab kit cost was due to the various IC chips listed in Table 3, along with several simpler chips used earlier in the course (inverter, AND, OR, XOR). Also included in this cost were LEDs (one connected by 330Ω resistor to each of the 4 accumulator output pins), DIP switches (used with 2.2kΩ resistors as input for various lab assignments), transistors (used only in the first lab), and various resistors and capacitors required for the power supply and clock circuits. Finally, $50 payed for a single classroom EEPROM programmer.

7. COURSE CONTENT

The course was built around a series of digital electronics lab assignments, listed in Table 5. In many of the labs, students were asked to use simpler circuit elements to build more complex ones. For example, students used flip-flops and logic gates to build a simple counter, thereby earning a counter chip for their kits. (Likewise, successful completion of the final lab earned a student the right to use a real computer to implement a simple virtual machine, assembler, etc.)

Table 5. Lab Assignment Sequence

Lab Assignment	Summary
5 Volts	build voltage regulator; learn to use breadboard, switches, LEDs
Transistors	build logic gates from transistors
NAND Gates	build logic gates from NANDs
Combinational Logic	build selector/adder from NOT/AND/OR
The Clock	build clock circuit
Finite State Machines	build FSMs from gates and flip-flops
Counter / ROM	use counter to cycle through ROM data (later served as PC and program storage)
ALU / Register	load/subtract input value from register (later serves as accumulator datapath)
RAM Datapath	add selector, flip-flops, and RAM
Control Logic	connect program counter, control ROM; set 4-bit op-code to test instructions
Program Execution	connect program ROM and clock; write/execute programs

The building of the processor was assigned incrementally, rather than as one large project. Thus, seven of the assignments listed in Table 5 walked the students through the building of a particular subsystem of the emerging processor. Although these labs provided students with the design (and sometimes explicitly told students which pins to use), the layout of chips on the board and much of the wiring decisions were left to the students.

The course also addressed a variety of conceptual material, including the static discipline, Karnaugh maps, finite state machines, and a cursory discussion of computability. In focusing only on material essential for teaching what a computer is, we omitted topics such as timing, pipelining, caching, interrupts, virtual memory, and operating systems. On the other hand, students would not truly understand what a computer is without understanding what it means for a machine or language to be

universal. Thus, the final two weeks of the course were devoted to an exploration of the power of the CHUMP language itself.

8. THE CHUMPANESE LANGUAGE

Clearly the CHUMP's 64-bit RAM is no match for a Turing machine's infinite tape. Therefore, we began talking about what the CHUMP language could do if it weren't limited by 4-bit values, addresses, etc. We referred to this unlimited form of the language as "Chumpanese", and represented it with an assembly-like syntax. In Chumpanese:

- Programs may be arbitrarily long.
- All values are integers of arbitrarily large magnitude.
- There are an infinite number of memory locations.
- Names are used to identify line numbers, memory addresses, and memory offsets.

For example, the following listings show a simple Chumpanese program and an equivalent Java program.

```
loop: READ count        while (true)
      LOAD IT             count++;
      ADD 1
      STORETO count
      GOTO loop
```

It can be shown that Chumpanese is indeed a universal language by demonstrating that any program in a known universal language can be translated into an equivalent Chumpanese program. (For example, the universal One Instruction Computer's `SUBZ` command can be translated into a sequence of nine Chumpanese instructions.) However, we decided that such a proof would be tedious and potentially unconvincing to our students. Instead, we aimed to develop their intuition that Chumpanese was universal by teaching them how to emulate the common programming constructs of a language they already believed to be universal: Java.

We therefore taught students to represent variables as memory addresses in Chumpanese, to translate `if` and `while` statements into sequences with `IFZERO` and `GOTO` commands, to see reference-type variables as memory locations containing addresses of other memory locations, to represent arrays and simple objects (really structs) as contiguous segments of memory, and to view array indices and instance variable names as offsets into that memory. We showed students how to represent a stack and how to write Chumpanese sequences that could push/pop stack values. Finally, we taught students to use `PUSH` and `POP` macros to implement procedures and procedure calls, thereby connecting their understanding of computer architecture to the material they had studied in AP Computer Science.

9. REFLECTIONS

On the whole, the course and the lab kit were overwhelming successes, with all 18 students building functional computer processors (with varying degrees of help). Students enjoyed the course quite a bit, and final exam responses indicate that most left with a thorough understanding of how to design combinational logic circuits and finite state machines, how their processor worked, and how to translate the simplest Java code segments into machine language.

Of course, there are a few aspects of the course that did not work out as well as we had hoped. In our software-based courses, students are able to debug most errors without teacher assistance. When a student does need help, a quick glance at their output or code is usually sufficient for us to advise them as to what to try next. Our experience teaching computer architecture, however, was entirely different. Although a few students did master the art of hardware debugging, many of our best programmers found themselves helpless to debug their circuitry. It seemed that students were constantly asking for help from every corner of the classroom. Because a hardware bug typically takes much more time to address than a software one, helping a single student could easily devour an entire class period, while other students grew increasingly frustrated waiting for help.

Although the processor design proved to be robust and reproducible, the students found some of the supporting elements of the lab kit to be frustrating at times. By far, the largest hardware frustration we faced concerned power consumption. Students spent hours debugging problems due only to dead batteries. Although the simple circuits students built at the beginning of the course did not draw much current, the completed CHUMP, with its EEPROMs, RAM, ALU, etc., drained the 9-volt batteries at a rate sometimes exceeding one battery per hour of use. We therefore needed to replace batteries frequently near the end of the course.

Finally, students had difficulty with the CHUMP instruction set. They frequently confused the `READ` and `LOAD` commands, as well as the constant and memory variants of each instruction. This confusion hampered their ability to debug their processors, although most students understood the instruction set by the time the course ended. In retrospect, we wish we had provided the students with a Chumpanese virtual machine, and had them use it to complete a set of programming exercises before assigning them to wire the CHUMP datapath.

It would also have helped to leave more time at the end of the course to explore more advanced Chumpanese programs involving procedures. These exercises proved to be too much for all students to grasp in a limited time. Nonetheless, the students' initial skepticism regarding the power of their simple processors gradually disappeared. Ultimately, we believe students left the course with an appreciation for how even the most high-level software must run as a sequence of simple commands by a computer processor which is no more than an arrangement of fluctuating voltages.

10. REFERENCES

[1] Lancaster, D. E. *TTL Cookbook*. Howard W. Sams, Indianapolis, IN, 1974.

[2] Madnick, Stuart. *Understanding the Computer (Little Man Computer)*. Unpublished manuscript, 1993.

[3] Patterson, D. A. and Hennessy, J. L. *Computer Organization and Design*. Morgan Kaufmann, San Francisco, CA, 2004.

[4] Ward, S. and Terman, C. *6.004 Computation Structures*. MIT OpenCourseWare, 2002