ROYAL ST. GEORGE'S COLLEGE

ADVANCED COMPUTER ENGINEERING SCHOOL

chump

CHEAP HOMEMADE UNDERSTANDABLE MINIMAL PROCESSOR

ICS4U-E students have earned the technical and scholastic skill set to permit them to engage the quintessential computer engineering project: **the 4-bit TTL Processor**. The purpose of this workbook is to support students' hands-on creation of a processor that will execute machine language instructions on device based on Harvard architecture.

ACE: ____

Course: ICS4U-E

Year: 2025-2026

Instructor: C. D'Arcy

Based on: *A Simple and Affordable TTL Processor for the Classroom* D. Feinberg. 2007.

Photo: D. Raymond & J. Corley's CHUMP. Spring 2019.
Video: https://www.youtube.com/watch?v=vZ13xud0qBc

Quote: *People who are serious about software should make their own hardware*. Alan Kay, July 1982.

Why are Sr. ACES Asked to Undertake this Project?

The answer to this question is rooted in your teacher's philosophy of scholastic enrichment. RSGC's ACES program is *optional*, so you have engaged its curriculum by *choice*. It is not every secondary school student that willingly embraces time-consuming, demanding curriculum that is not required by any university admissions department. At this most critical stage of your young scholastic journey you have taken a leap of faith based on the expectation that you will be advantaged in the future for doing so. For the engineering-bound among you, this is a wise decision.

Chump Build. Photo courtesy of J. Vretenar ACES, Fall 2020

To truly understand and appreciate computer technology, or any sufficient complex system for that matter, **takes time**. You may not appreciate that you have more time on your hands now than you will as you get older. So, it is both a great privilege and a responsibility to determine how best to invest my students' time for maximum future advantage.

Through feedback from ACES alumni, I have a good sense of what lies in wait for you in an undergraduate engineering program. Although I receive no shortage of suggestions from them on what specific facts and techniques the universities had taught them, our program is less about saving you from having to go to class, and more about giving you the perspective you need to engage new material you are expected to master. For example, when your project group is meeting to discuss the sequence of steps required to get a prototype across a finish line in four weeks, your experience and skills will temper their fantasies about a hassle-free implementation and suggest a more realistic course of action.

ACES is an acronym for Advanced Computer Engineering School in which we engage projects within its realm. Building a processor from TTL chips is the quintessential 'rite-of-passage' in preparation for post-secondary courses in computer architecture.

Apart from the deep knowledge and insights you will gain, the time-consuming, challenging, and frustrating days and weeks of building, wiring, rewiring, and debugging are all conditioning you for what lies ahead. Few projects you have undertaken compare to the complexities of building a working CHUMP and conditioning yourself to its uphill trajectory and supportive time-management and resilience skills will test the depth of commitment to an undergraduate engineering program.

Table of Contents

Feedback from ACES' Alumni

"*The freedom to design, build, make mistakes and troubleshoot that your courses allow is incredibly important as when you get to university your freedom to do those things are severely limited in place of more standard textbook & PowerPoint based courses. The ISPs that I worked on at RSGC (Line & Wall Avoiding Robots) were a big factor in my decision to specialize into the mechatronics following the general engineering first year. Without that experience, picking a stream of engineering would have been much foggier*." **R. Fletcher, Mechatronics Engineer, ACES'15, Western '20**

"*I've found a new appreciation for my three years in the ACES program. I've talked with my housemates who came from different schools across Toronto and they've admitted, after looking ahead at the course's later content, they are a little bit worried. This is where my new found appreciation comes in. The vast majority of the courses' content has been covered in either the Grade 11 or Grade 12 ACES classes and because of that, I've been given a very sizable advantage before the course has really even started. In addition to the design, software, and hardware knowledge I gained from the program, the ability to write well thought out, concise, neatly formatted, and very visually appealing reports in a short amount of time has been paramount.*" **J. Lank, Engineering, ACES'20, Queen's '24**

S.Atkinson•P.Bizzari•A.Goldman
•R.McKenna•L.Robertson-Caryl•
J.Shibley•J.Vretenar•C.D'Arcy
ENGINEERING

"*My ACES knowledge continues to make my life relatively easy at school. I finished up the second Digital Circuits course with a strong mark. I am taking Computer Organization this semester which is about half MIPS 32 based Assembly programming, and the other half basic computer structure (lots of similarities to CHUMP here). It's great to see some of this stuff again - while a bit repetitive at times it's nice to have at least one course where you can count on your own brain for the content.*"

G. Benson, Commerce and Engineering, ACES '19, U of Calgary '23

"*I thought this would be a good time to say how grateful I am for everything I learned in your class. It continues to pay dividends in everything I work on both inside and outside of school.*"

E. Peterson, ECE (Direct Entry), ACES '18, Queen's '22

"*As I continue with my job search for winter term, one of the jobs I came across was as an assistant in the MME/WatIMake clinic, this job would entail consulting and aiding students/faculty with rapid prototyping, machining, and embedded software development, basically TA 2.0. I was the only 1st year to be selected for an interview, amongst 2nd, 3rd, and even 4th years. While I did not get the job, the interviewer did go out of their way to tell me that I had a huge capacity for all aspects of engineering, much more than was expected of a 1st year.*"

J. Dolgin, Engineering, ACES '20, Waterloo '25

"*Yesterday I had an interview with AMD for an internship position. In the interview they asked me extensively about the projects I worked on in the ACES program. Today, I was offered the job there. I have no doubt that your course was the primary reason I was offered the job, and I am reminded of how important it is to my life. I want to thank you for the time you invested in me as a student*."

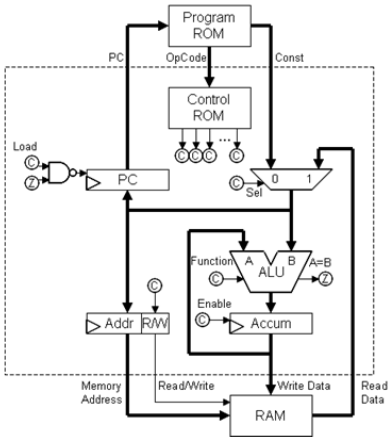
O. Logush, Electrical Engineering, ACES '18, Queen's '23

0 Acknowledgements

Dave Feinberg, B. Sc., M. Eng.

This MIT graduate found himself teaching Computer Science at the Harker School in California in 2007. To round out his students' Grade 11 AP CS experience he elected to introduce them to hardware in Grade 12 by asking them to build a breadboard processor. His pdf guide entitled *A Simple and Affordable TTL Processor for the Classroom* found its way onto the internet and eventually into our hands in 2011. Feinberg's design of a 4-bit, TTL level processor is modelled after a similar computational platform based on a Harvard architecture of a device he explored while at MIT. As far as I can tell, Feinberg left the Harker School around 2010 and moved to Ohio where he now teaches at the Columbus Academy, a private country day school. https://tinyurl.com/smpmd9be

What started out as a curiosity for some of the ambitious ACES in *any* grade in the ensuing decade has evolved into cornerstone of our Grade 12 course. Few activities can match the numerous benefits the complete build provides, not the least of which to is set the stage for the Assembly language exploration of AVR 8-bit microcontrollers in the latter half of the senior course.



Indeed, ACES have gained so much advantage from Feinberg's masterpiece I felt the need to reach out, thank him personally, and share some of the heights ACES have taken his CHUMP project to. In a recent email exchange, Feinberg shared this response,

"*This past semester I taught my course over Zoom, and so I couldn't have the kids work with actual hardware. Instead, I used CircuitVerse--a free digital electronics simulation website. It's sometimes glitchy, but mostly I was very impressed by what it could do (and how easy it was to use). After the students completed labs on combinational logic and finite state machines, I had them work through this tutorial on the Chumpanese language.*"
https://tinyurl.com/5yrv3e5z

"*They entered their programs into this simple virtual machine:*"
https://tinyurl.com/t9jayhw7 (*may have to type in*)

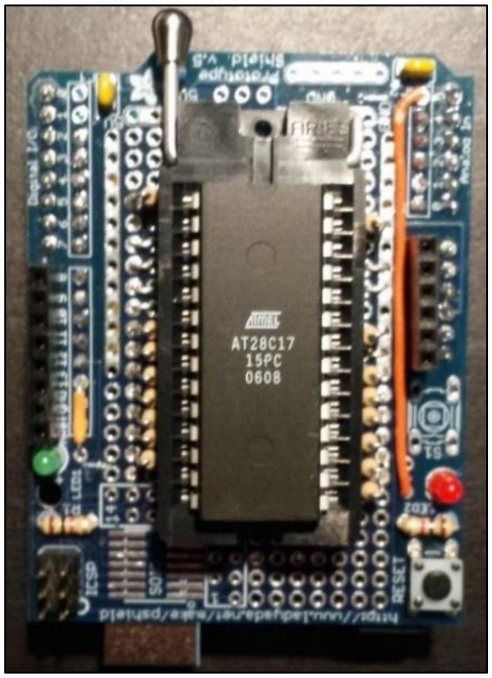
"*Then students learned about the CHUMP processor by completing this lab, which walked them through all the parts of the CHUMP and had them fill in ROM values to make it work.*"
https://tinyurl.com/7a7rrjv6

Finally, I could be mistaken, but I believe that after a decade, and reviewing what ACES have done with his outline, Feinberg was inspired to take a fresh look at his former brilliance. The virtual simulations are the result and, as far as I know, he has resumed presentation of this curriculum to his computer science classes at Columbus Academy.

ACES' Chump Legacies and Archives

2015/2016 Jackson Russett

While he was a Grade 10 student, Jackson approached me with a copy of Feinberg's paper and asked if I'd help him go through it, with a view to building the full processor. Our half-course ended before Jackson could finish the full build but not before he successfully developed an Arduino Shield that he used to flash the AT28C17 EEPROM (right).



2017/2018 Ethan McAuliffe

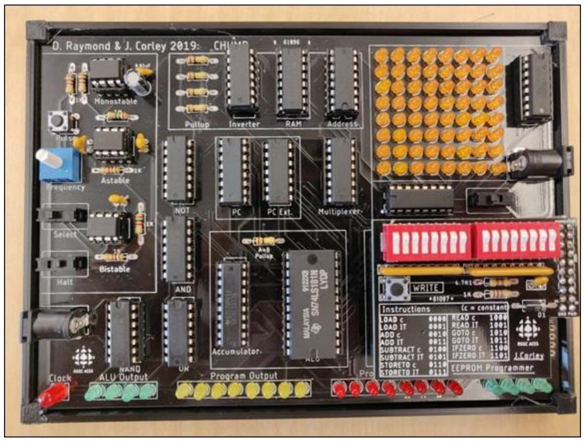
The 2017/2018 ICS4U class was the first group tasked with developing the CHUMP. Although no one completed the build in full, Ethan took on the challenge of building an 8-bit processor based on Ben Eater's design for his Long ISP. This proved too great a challenge to complete in the 8 weeks that were budgeted. Not to be defeated, Ethan returned to the 4-bit design over the summer of 2018 and was triumphant. Read all about it at: https://tinyurl.com/nj8bpx8t

2018/2019 Dan Raymond and James Corley

In the 2018/2019 ICS4U class was given their crack at the CHUMP. Dan and James engineered a successful build in the Fall. Having enjoyed the experience, they requested the build for their Long ISP in which they would produce a custom PCB and EEPROM burner plugin. They did exactly that as this video describes: https://tinyurl.com/rwm4bvsu

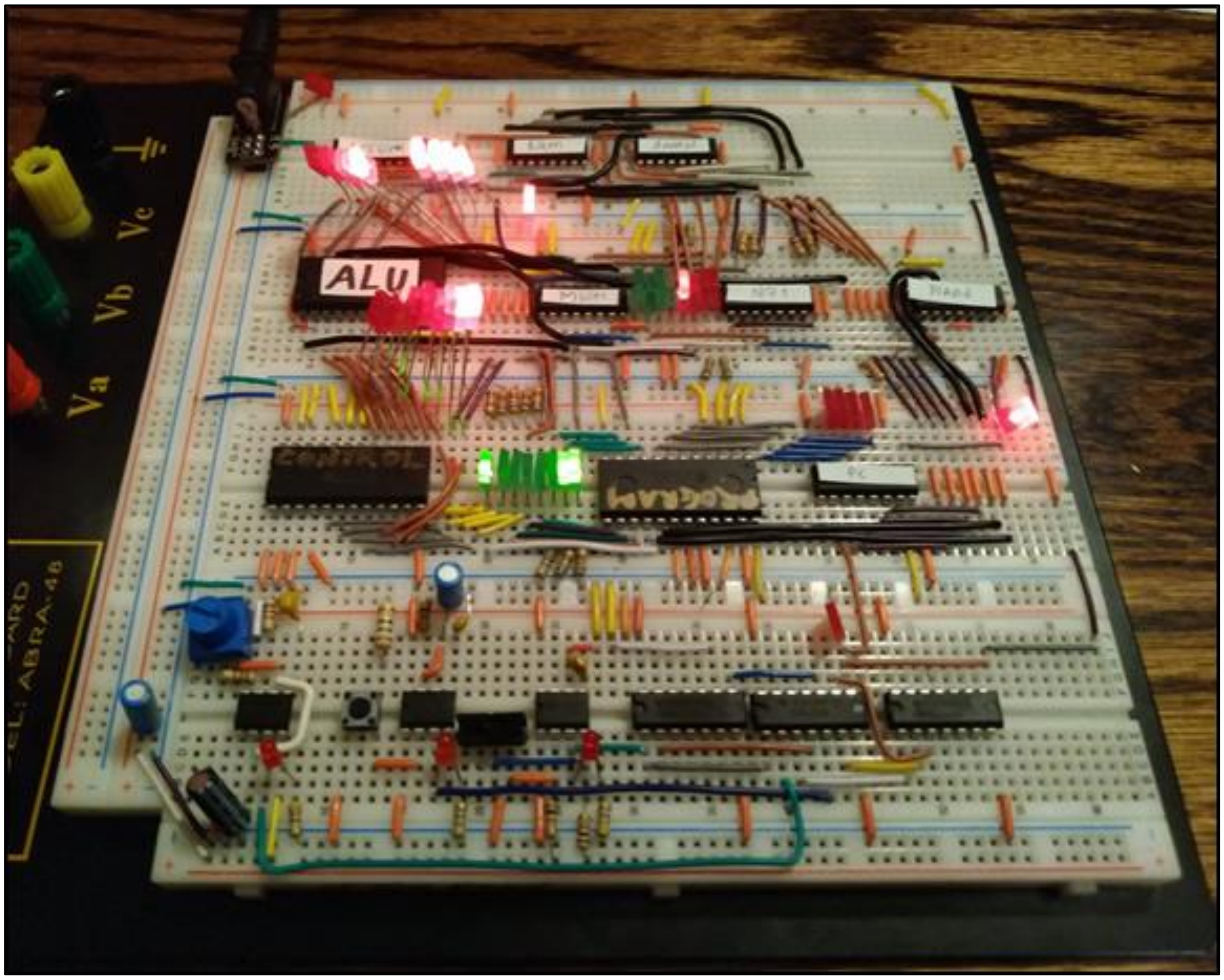
Raymond CHUMP without Corley Programmer

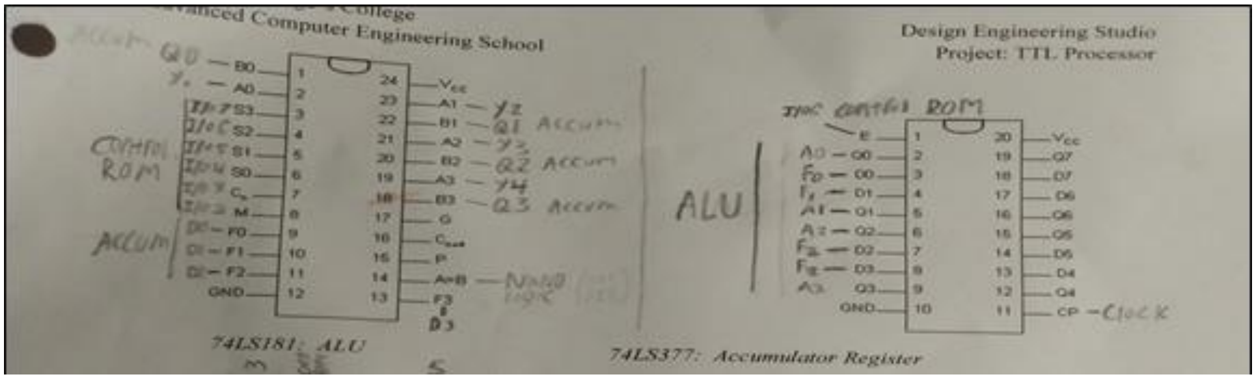
Raymond CHUMP with Corley Programmer



2019/2020 Josh Dolgin and Max McCutcheon

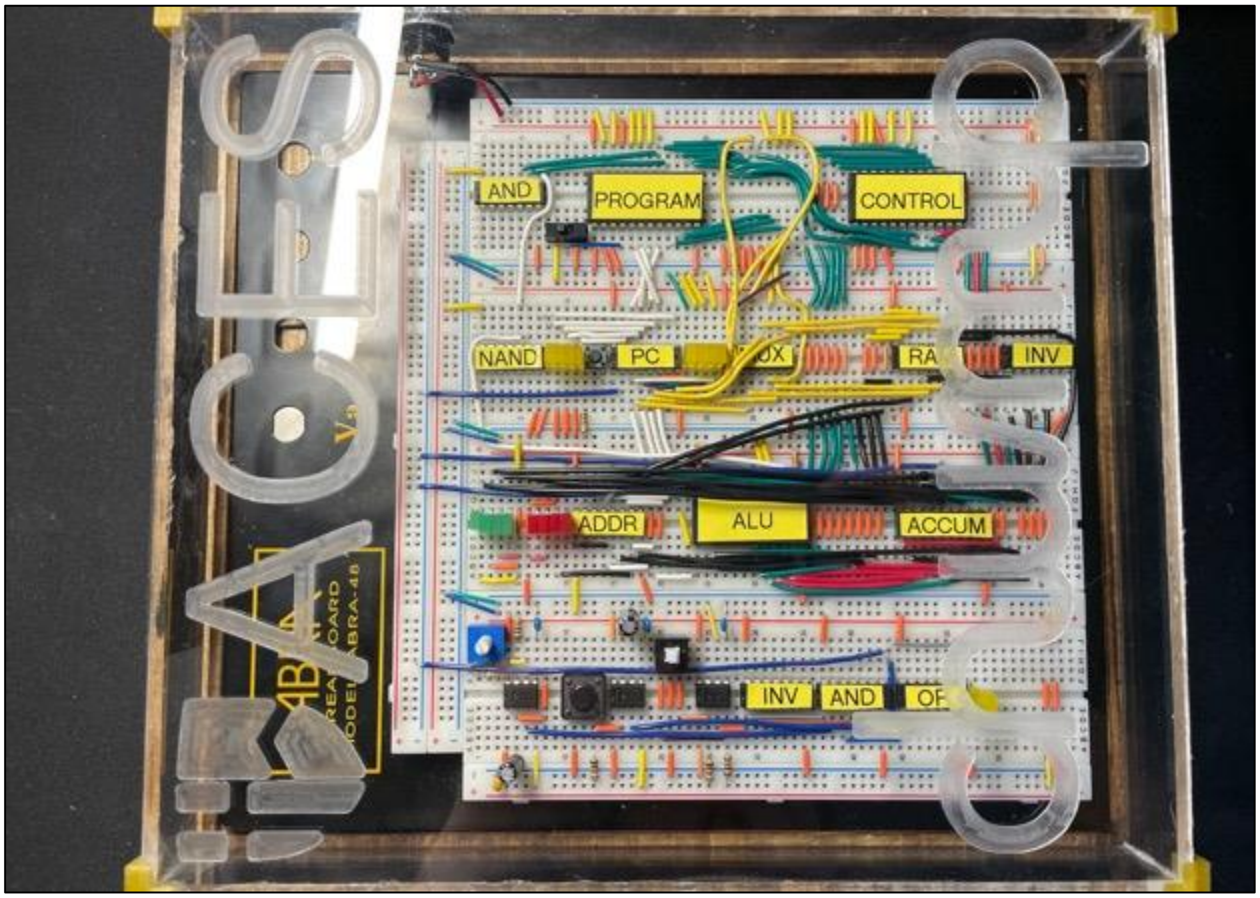
By the third year, Sr. ACES had had both the benefit of the two previous' years progress and the confidence that the summit of the project was attainable. Despite weeks of arduous research, wiring, flashing of EEPROMs and debugging, many of our intrepid Sr. ACES eventually completed their marathons. Two leaders within the group, Josh Dolgin and Max McCutcheon, shared their daily progress with their peers along the way and deserve special mention. Max's build and DER notes appears below. Here is Josh's CHUMP video summary: https://tinyurl.com/b2kbpfu4





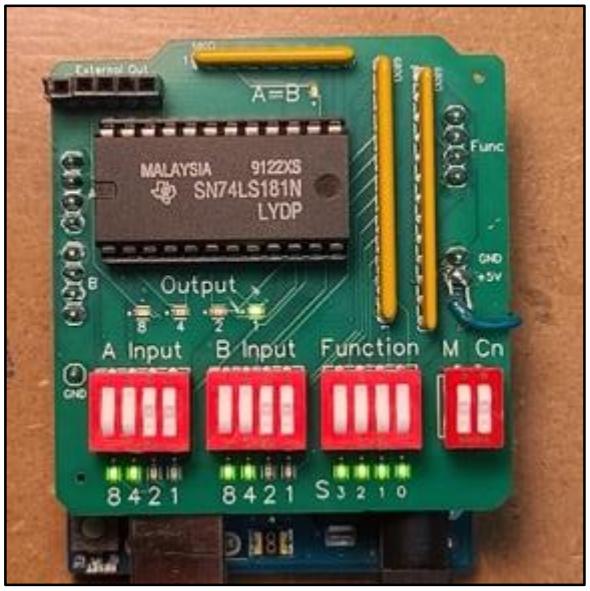
2020/2021 Joseph Vretenar

On the strength of the three previous build years, the 2020/2021 Sr. ACES were well positioned to tackle their projects. An unanticipated advantage turned this year into a wildly successful iteration as the mutual support these ACES offered each other was a joy to behold. For his Short ISP Joseph was excited to put his newly-acquired CNC machine to use by milling out a custom-designed wooden CHUMP case with an acrylic top. https://tinyurl.com/yrr2saxe



S.Atkinson•P.Bizzari•A.Goldman
•R.McKenna•L.Robertson-Caryl•
J.Shibley•J.Vretenar•C.D'Arcy
NGINEERIN

2020/2021 Liam Roberston-Caryl



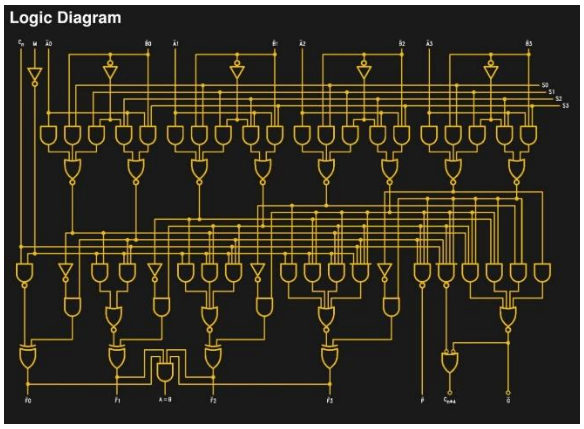
The computational performance of the CHUMP falls on the shoulders of the SN74LS181 *Arithmetic and Logic Unit* from Texas Instruments. A good portion of the workbook text to follow covers the features (*and mysteries*) of this historically significant IC.

Intrigued by the importance and idiosyncrasies of this chip, Liam Robertson-Caryll elected to commit his Short ISP selection to the design and implementation of a UNO Shield that could be exploited to demystify the device.

Two trips to JLCPBCs in the Fall of 2020 produced a remarkably proficient solution that Liam demonstrates admirable in his summary video, https://tinyurl.com/468su2b

In his Reflection, Liam includes the following perspective, *"…I am glad that I chose the ALU Shield for my ISP because it both prepared me for the CHUMP and reinvigorated my passion for hardware design. In all, I got to feel what it is like to engineer something from an old datasheet to a modern PCB.*"

2020/2021 Jackson Shibley



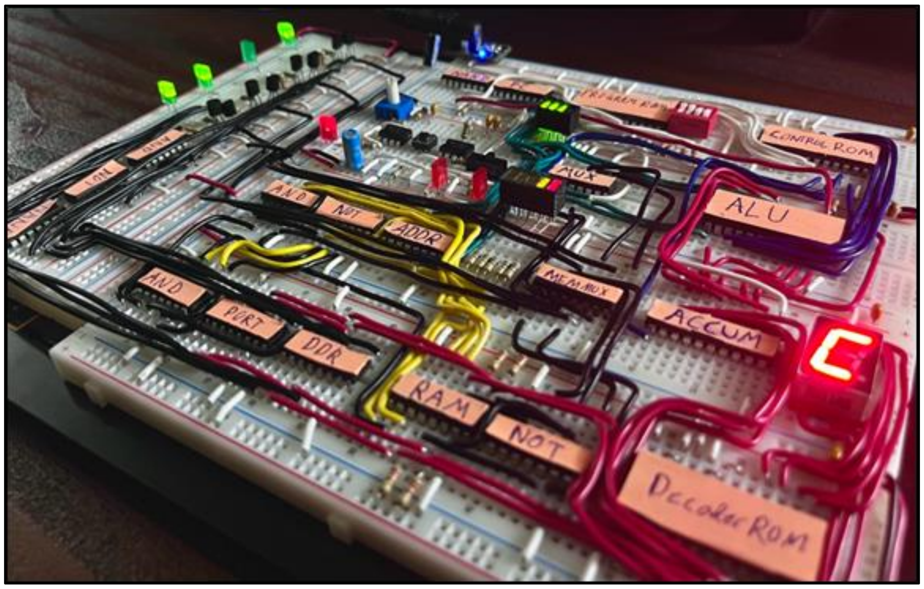
Jackson was drawn to the SN74LS181 *Arithmetic and Logic Unit* like a magnet. His long-standing passion for 20th Century legacy hardware provided all the incentive he needed to pull back the curtains on this complex device to expose the combinational logic design that drove its 1970s success.

Jackson's ALU emulator, which he dubbed the **ShiB181**, consisted of two boards. The **motherboard** implementation of the adjacent schematic included two dozen discrete logic ICs and headers for connectivity. A **daughterboard** attached to the motherboard provides visual confirmation of the operands, function selection, and control lines. Here is Jackson's summary video, https://tinyurl.com/3rz5weuz

In his Reflection, Jackson relates the following perspective, "*When I started to investigate the SN74LS181 in July 2020, I was immediately intrigued by the efficiency of its logic design. As I continued to study the chip, I began to learn more than I ever could have imagined. I learned about complex Boolean arithmetic, propagation delay, ripple counting, and much more. By the end of the summer, I decided that I had learned enough and I should put all of my newfound knowledge into practice.*"

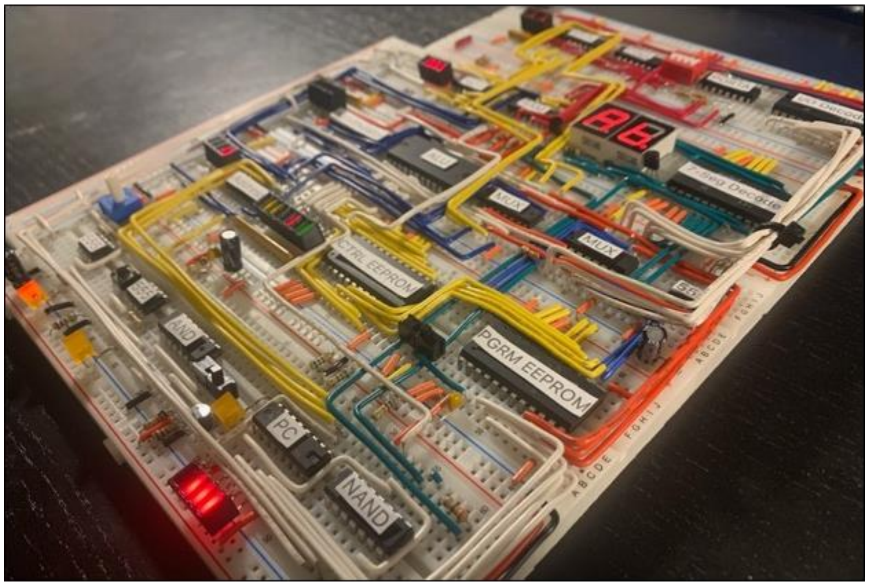
2023/2024 Graham Davidge

"*This project was potentially the coolest and most interesting thing I have ever done. From reading about CHUMP at the beginning of the term, to watching my completed CHUMP work, I was engaged by this project.*" https://www.youtube.com/watch?v=yLP8whVingA



2023/2024 Liam McCartney

"*This project was a ton of fun to build. The room for customization within CHUMP is enormous, and gives room to branch out into anything you would like. No two CHUMPs will end up looking alike. The other thing I liked about CHUMP is how the group comes together to get it done. We all had our own chip assignments, and we taught and learned with each other.*"
https://www.youtube.com/watch?v=DcIRDraD-Dk

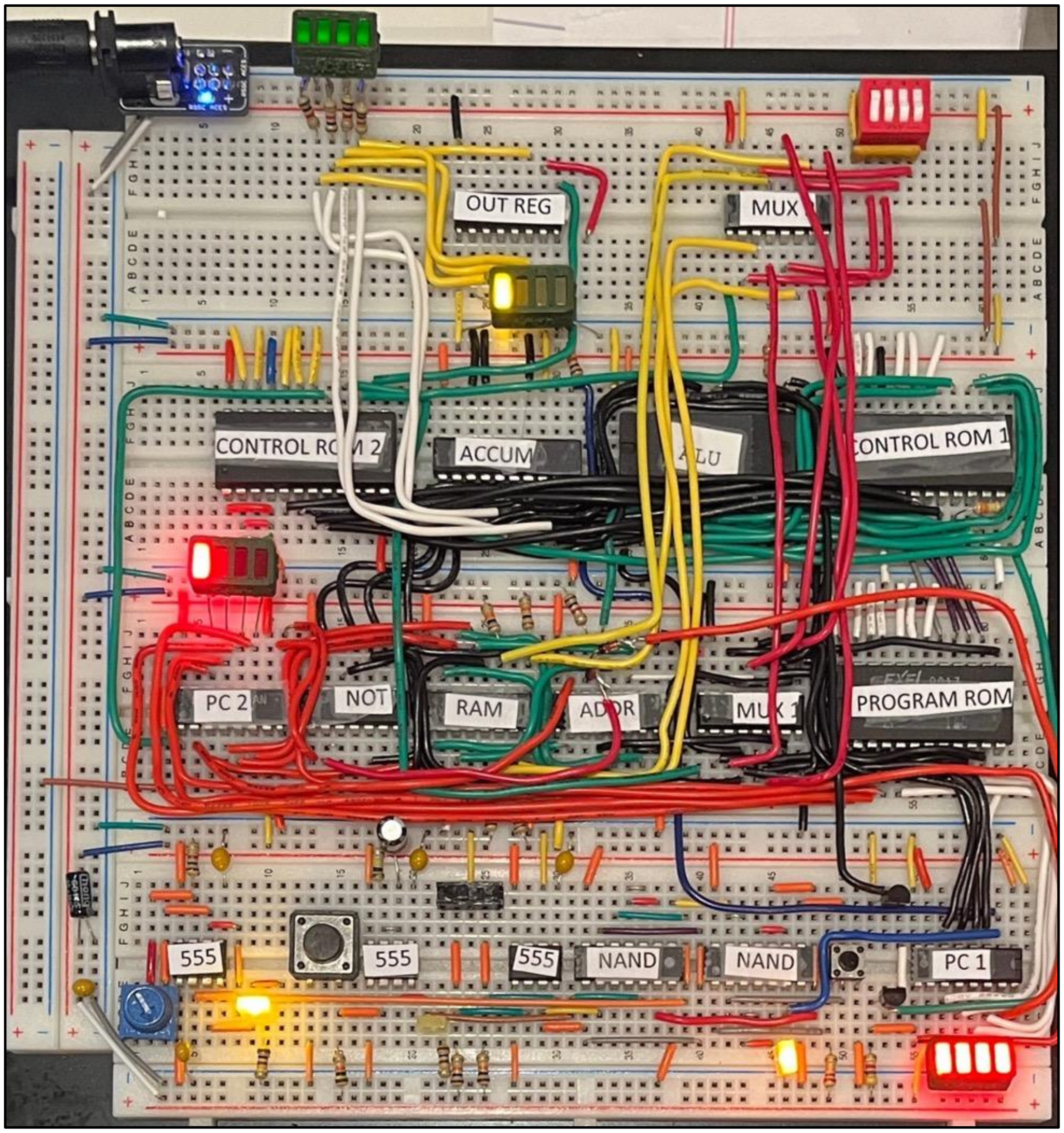


2024/2025 Rohan Jamal

"This has been a phenomenal project. I can honestly say that I genuinely understand the entire CHUMP, and I fulfilled all of my wishes from the start of the project. Over the last month, many days were spent in the DES until 6 or 7 P.M. I was fascinated from the start with increasing the clock frequency from 1 Hz to a much higher frequency. I managed to get the frequency to 90.23 KHz, and in a way that my program did genuinely benefit from the additional processing power.

This project also answered one of my biggest questions about microcontrollers and computers (Arduinos in particular): last year, I remember asking Mr. D'Arcy how the Arduino IDE transfers the code you write onto the chip itself and how the processor is able to interpret the code. Now that I have written my own assembly code, converted it to machine language, and built the processor for it to run on, I think that I do understand how it works.

https://www.youtube.com/watch?v=xfGtRVXh8Xc



Ben Eater

ACES are already familiar with Ben Eater's remarkably informative and insightful YouTube videos on electronics. Here is the link to his Channel:

https://tinyurl.com/jnq3hrg

Over the past couple of years Eater has followed a path in parallel with ACES in which he outlines, in wonderful detail, the development of an 8-bit 6502-based computer on a breadboard. Furthermore, all the parts can be purchased in kit form from his web shop at,

https://eater.net/shop

Ironically, it is his own sentiments expressed in the first 30 seconds of his must-see *Why Breadboards?* video [https://tinyurl.com/vzzfu95] that steers ACES away from Eater's design and towards CHUMP. Apart from the fact that the time-savings of a 4-bit design in the short timeframe our course imposes is significant, his explanations are so good that it turns his 6502 design into a paint-by-numbers project that leaves little room for ACES to pursue the real benefit of an exercise like this: troubleshooting. With little more than the description this workbook provides, to a large extent you are, purposely, on your own.

Eater Video Gallery

There are a number of videos that Eater offers that are useful for our CHUMP processor.

Why Breadboards? https://www.youtube.com/watch?v=fCbAafKLqC8

555 Clock Series

Astable 555 timer - 8-bit computer clock - part 1
https://www.youtube.com/watch?v=kRlSFm519Bo

Monostable 555 timer - 8-bit computer clock - part 2
https://www.youtube.com/watch?v=81BgFhm2vz8

Bistable 555 - 8-bit computer clock - part 3
https://www.youtube.com/watch?v=WCwJNnx36Rk

Clock logic - 8-bit computer clock - part 4
https://www.youtube.com/watch?v=SmQ5K7UQPMM

Loops in Assembly https://www.youtube.com/watch?v=ZYJIakkcLYw

Build an EEPROM Programmer
https://www.youtube.com/watch?v=K88pgWhEb1M

1 Software

You are all but guaranteed *never* to be in this position again; that is, to write software for a processor that you *intend* to build. In learning how to code for a processor that does not exist, you are creating a sense of mystery, suspending the instant gratification of today's programming environments in favour of patience, apprehension, and, ultimately, a feeling of the deepest sense of accomplishment.

We will begin with the language of the CHUMP or 'CHUMP*anese*' as Feinberg called it.

1.1 Instruction Set

The 4-bit parallel address, control and data buses of our TTL processor limits our design to 2^4 or 16 values for any given interpretation, including the instruction set. The table below list two pairs of the seven standard instructions in the CHUMP definition. The two variations of an eighth instruction are meant to be user-defined. Incidentally, this is not exactly the original specification defined by Feinberg. ACES have discovered a hardware advantage in tweaking the opcodes slightly. This is one of the unexpected dividends of a rich computer engineering project like this one in which a deep understanding of the hardware informs optimum software architecture.

OpCode (Machine)	Operand (const/mem)	Mnemonic (Assembly)	Summary
0000	const	LOAD	accum ← const; pc++
0001	mem		accum ← [addr]; pc++
0010	const	ADD	accum ← accum + const; pc++
0011	mem		accum ← accum + [addr]; pc++
0100	const	SUBTRACT	accum ← accum - const; pc++
0101	mem		accum ← accum -[addr]; pc++
0110	const	STORETO	[const] ← accum; pc++
0111	mem		[addr] ← accum; pc++
1000	const	READ	addr ← const; pc++
1001	mem		addr ← [addr]; pc++
1010	const	USER	?
1011	mem		?
1100	const	GOTO	pc ← const; pc++
1101	mem		pc ← [addr]; pc++
1110	const	IFZERO	accum==0? pc ← const : pc++
1111	mem		accum==0? pc ← [addr] : pc++

1.2 Program Structure and Storage

First of all, since there is no IDE, CHUMP programs are written either by hand with pencil and paper, or with the help of a text editor. Word, Excel, TextEdit, Notepad or even the Arduino IDE, all work well.

A CHUMP **instruction** consists of an 8-bit binary value (*one byte*). The leftmost 4-bits (*high nibble*) form the Operation Code (*OpCode*) and the rightmost 4-bits (*low nibble*) provide the operand (*a value interpreted as either a literal constant or the address of a RAM location*). The distinction between the two is made by the LSB of the Opcode (0-*constant*, 1-*address of a RAM location*).

A CHUMP **program** consists of a *maximum* of 16 instructions. This limitation is a direct consequence of the 4-bit address bus.

Programs are flashed onto your 8-bit parallel EEPROM IC through the use of the ACES' EEPROM Burner Shield that has been lent to you. Additional details supporting its use are provided in a later section.

The EEPROM address of an instruction is that instruction's de facto **line number**. Line numbers can range from 0000 to 1111. Line numbers also serve as targets for the two branch instructions.

1.3 Instruction Examples

1.3.1 ADD Instruction

The instruction below was written into address 7 of the Program EEPROM. When executed, the ALU will add 2 to the contents of the Accumulator. The contents of the Program Counter will be increased by 1. The next instruction to be executed is in Program EEPROM address 8.

Line Number	Instruction	Mnemonic	Operand
`0111`	`00100010`	`ADD`	`2`

1.3.2 IFZERO Instruction

The instruction below was written into address 2 of the Program EEPROM. If the Accumulator contains the value of 0, the Program Counter is loaded with a value of 6. The next instruction to be executed is in Program EEPROM address 6.

Line Number	Instruction	Mnemonic	Operand
`0010`	`11100110`	`IFZERO`	`6`

1.3.3 GOTO Instruction

The instruction below was written into address 15 of the Program EEPROM. The Program Counter is loaded with a value of 0. The next instruction to be executed is in Program EEPROM address 0.

Line Number	Instruction	Mnemonic	Operand
`1111`	`11000000`	`GOTO`	`0`

1.3.4 LOAD Instruction

The instruction below was written into address 10 of the Program EEPROM. This instruction expects to have been preceded by a READ instruction which placed a 4-bit binary value in the Address register (call it `aaaa`). This instruction loads the contents of RAM Address `aaaa` into the Accumulator. The Program Counter is increased by 1. The next instruction to be executed is in Program EEPROM address 11.

Line Number	Instruction	Mnemonic	Operand
`1010`	`00010000`	`LOAD`	`0`

1.4 Program Guidelines

1. CHUMPanese programs are typically developed using the assembly mnemonics with either numeric operands (in decimal) for *constants*, or the generic `IT` for *memory* instructions. We refer to this form of CHUMPanese as **Assembly Language**. When your program is ready for flashing into the Program EEPROM, it will have to be first translated into its 8-bit binary form. We refer to this form of CHUMPanese as **Machine Language**.

2. **Each** assembly statement should be followed by a short, meaningful comment. The convention is to start a comment with a semicolon.

3. Every `READ` instruction must be followed by memory instruction; in which `IT` stands as a placeholder for the operand. Conversely, every memory instruction must be preceded by a `READ` instruction. This, then, is the general Assembly sequence for RAM access,

```
READ  __

_____  IT
```

4. Your assembly code can make use of *variables*. A *variable* is simply a name for the RAM address where its value is stored. You may wish to maintain a separate table or map for variables and their corresponding RAM Address.

5. It takes two instructions to load (→`ACCUM`) the value of a *variable*.

```
READ x     ;place x in the Address register
LOAD IT    ;place contents of RAM address x in the Accumulator
```

6. It takes only one instruction to write to a variable (→`mem[x]`).

```
STORETO x  ;place contents of Accumulator in RAM address x
```

7. Feinberg has created a wonderful introductory tutorial on the CHUMP*anese* language and has created a virtual CHUMP emulator to enable users to test drive their programs by stepping through each instruction and monitoring the accumulator. This is ideal preparation for our exploration of AVR Assembly language later in the year in Atmel Studio 7.

   a) Open the CHUMP virtual machine at https://codepen.io/davefdavef/full/WNxRpMR

   b) Open Feinberg's tutorial and follow the instructions, https://tinyurl.com/3sxjdrma

   c) Review Feinberg's CHUMP Lab, https://tinyurl.com/7a7rrjv6

1.5 Common Program Structures

1.5.1 Swapping Variables

Swapping the contents of two RAM locations, say `x` and `y`, in assembly, can be written as follows. (assume `x` is 5, `y` is 10, and `temp` is 15; `[n]` means the contents of RAM Address `n`)

High Level	Machine Level		CHUMP (Assembly) Level	Comment
	Address	Instruction		
`temp = x`	0000	**1000** 0101	**READ** x	addr←5, pc++
	0001	**0001** 0000	**LOAD** IT	accum←[5], pc++
	0010	**0110** 1111	**STORETO** temp	[15]←accum, pc++
`x = y;`	0011	**1000** 1010	**READ** y	addr←10, pc++
	0100	**0001** 0000	**LOAD** IT	accum←[10], pc++
	0101	**0110** 0101	**STORETO** x	[5]←accum, pc++
`y = temp;`	0110	**1000** 1111	**READ** temp	addr←15, pc++
	0111	**0001** 0000	**LOAD** IT	accum←[15], pc++
	1000	**0110** 1010	**STORETO** y	[10]←accum, pc++

1.5.2 *if/else* Statement

An Assembly `if/else` decision sequence can be written as follows.

```
           <some instruction>    ;the Accumulator has some value
           IFZERO zerocase       ;is the Accumulator 0?
           <nonzero case>        ;if not, continue execution here
           GOTO after            ;bypass the next fragment
zerocase:  <zero case>           ;execution begins here for 0
after:     <continue execution>  ;if/else is complete, keep going
```

In this example, if `x` is `3`, then `y` is assigned a value of `1`, otherwise `y` is assigned a value of `2`. Assume `x` is RAM Address 5; `y` is RAM Address 10; `[n]` means the contents of RAM Address `n`.

High Level	Machine Level		CHUMP (Assembly) Level	Comment
	Address	Instruction		
`if (x == 3){`	0000	**1000** 0101	**READ** x	addr←5, pc++
	0001	**0001** 0000	**LOAD** IT	accum←[5], pc++
	0010	**0100** 0011	**SUBTRACT** 3	accum←accum-3, pc++
	0011	**1100** 0111	**IFZERO** 7	accum==0?pc←7:pc++
`y = 1;}`	0100	**0000** 0010	**LOAD** 2	accum←2, pc++
	0101	**0110** 1010	**STORETO** y	[10]←accum, pc++
`else {`	0110	**1010** 1001	**GOTO** 9	pc←9
`y = 2;}`	0111	**0000** 0001	**LOAD** 1	accum←2, pc++
	1000	**0110** 1010	**STORETO** y	[10]←accum, pc++
	1001	**1010** 1001	**GOTO** 9	pc←9 ;∞ loop!

1.5.3 *`while`* Loop

An Assembly `while` iterative loop (with a `!=` condition) can be written as follows.

```
           <some instruction>    ;the Accumulator has some value
loop:      IFZERO after          ;if Accumulator is 0, end loop
           <loop body>           ;one or more statements in loop
           GOTO loop             ;return to top of loop and test
after:     <continue execution>  ;loop complete, keep going…
```

In this example, `x` has some value (it was explicitly assigned a value of `3` for illustration). The loop continues as long as `x` is not `0`. The sample loop body simply decrements `x`. Assume `x` is RAM Address 5; `y` is RAM Address 10; `[n]` means the contents of RAM Address `n`.

High Level	Machine Level		CHUMP (Assembly) Level	Comment
	Address	Instruction		
`x = 3;`	0000	**0000** 0011	**LOAD** 3	accum←3, pc++
	0001	**0110** 0101	**STORETO** x	[5]←3, pc++
`while(x != 0)`	0010	**1110** 0110	**IFZERO** 6	accum==0?pc←6:pc++
`x--;`	0011	**0100** 0001	**SUBTRACT** 1	accum←accum-1, pc++
	0100	**0110** 0101	**STORETO** x	[5]←accum, pc++
	0101	**1100** 0010	**GOTO** 2	pc←2
	0110	**1100** 0110	**GOTO** 6	pc←6 ;∞ loop!

1.5.4 Array Update

A high-level array assignment statement may appear as `a[i]=x;` where `a` is the array, `i` is the index within the array and `x` the new contents of array element at `i`.

For the sake of this example, we'll assume we wish to execute the specific assignment statement, `a[2]=7`.

We'll arbitrarily declare `a` to be an array of length 5, with a base address in RAM of 10.

The goal is to assign an `x` value of 7 to `a[2]`, which would imply a target RAM address of 12.

The CHUMPanese code below first initializes the variables, prior to determining the target address and finally, the assignment.

Symbol Table		RAM Contents	
Variable	Address	Before	After
	0000		
	0001		
	0010		
a	0011		10
i	0100		2
x	0101		7
temp	0110		12
	0111		
	1000		
	1001		
	1010		
	1011		
	1100		7
	1101		
	1110		
	1111		

High Level	Machine Level		CHUMP (Assembly) Level		Comment
	Address	Instruction			
i = 2;	0000	**0000** 0010	**LOAD**	2	accum ← 2, pc++
	0001	**0110** 0100	**STORETO**	i	[4] ← 2, pc++
x = 7;	0010	**0000** 0111	**LOAD**	7	accum←7, pc++
	0011	**0110** 0101	**STORETO**	x	[5] ← 7, pc++
a = 10;	0100	**0000** 1010	**LOAD**	10	accum ← 10, pc++
	0101	**0110** 0011	**STORETO**	a	[3] ← 10, pc++
;determine	0110	**1000** 0100	**READ**	i	addr ← 4, pc++
;the target	0111	**0011** 0000	**ADD**	IT	accum += [4](12),pc++
;address	1000	**1000** 0110	**READ**	temp	addr ← 6, pc++
;within	1001	**0111** 0000	**STORETO**	IT	[6] ← 12, pc++
;the array	1010	**1000** 0101	**READ**	x	addr ← 5, pc++
	1011	**0001** 0000	**LOAD**	IT	accum ← 7, pc++
	1100	**1000** 0110	**READ**	temp	addr ← 6, pc++
	1101	**1001** 0000	**READ**	IT	addr ← [6], pc++
	1110	**0111** 0000	**STORETO**	IT	[12] ← 7, pc++
;pause	1111	**1100** 1111	**GOTO**	15	pc = 15 ;∞ loop!

Note. It should be clear by now that a single high-level statement typically leads to more than one low-level assembly language instruction.

1.6 Sample Programs

1.6.1 Feinberg's Classic (Beginner)

Feinberg offers the program below in his original paper. Add high-level language statements and comments in the columns reserved in the style of the previous examples.

High Level	Machine Level		CHUMP (Assembly) Level		Comment
	Address	Instruction			
top:	0000	**1000** 0010	**READ**	2	
	0001	**0001** 0000	**LOAD**	IT	
	0010	**0010** 0001	**ADD**	1	
	0011	**0110** 0010	**STORETO**	2	
	0100	**1100** 0000	**GOTO**	top	

Write a short description of what this program does.

1.6.2 Shift Left (Intermediate)

Complete the three missing columns in the conventional style.

High Level	Machine Level		CHUMP (Assembly) Level		Comment
	Address	Instruction			
	0000		**LOAD**	0	
	0001		**READ**	y	
	0010		**STORETO**	IT	
	0011		**READ**	x	
	0100		**LOAD**	IT	
next:	0101		**IFZERO**	finish	
	0110		**LOAD**	2	
	0111		**READ**	y	
	1000		**ADD**	IT	
	1001		**STORETO**	IT	
	1010		**READ**	x	
	1011		**LOAD**	IT	
	1100		**SUBTRACT**	1	
	1101		**GOTO**	next	
finish:	1110		**GOTO**	finish	

Write a short description of what this program does.

1.6.3 Fill (Advanced)

In this final example, you are asked to complete the three missing columns in the conventional style.

High Level	Machine Level		CHUMP (Assembly) Level		Comment
	Address	Instruction			
	0000		**LOAD**	0	
	0001		**READ**	0	
next:	0010		**STORETO**	IT	
	0011		**READ**	IT	
	0100		**ADD**	1	
	0101		**GOTO**	next	

Write a short description of what this program does.

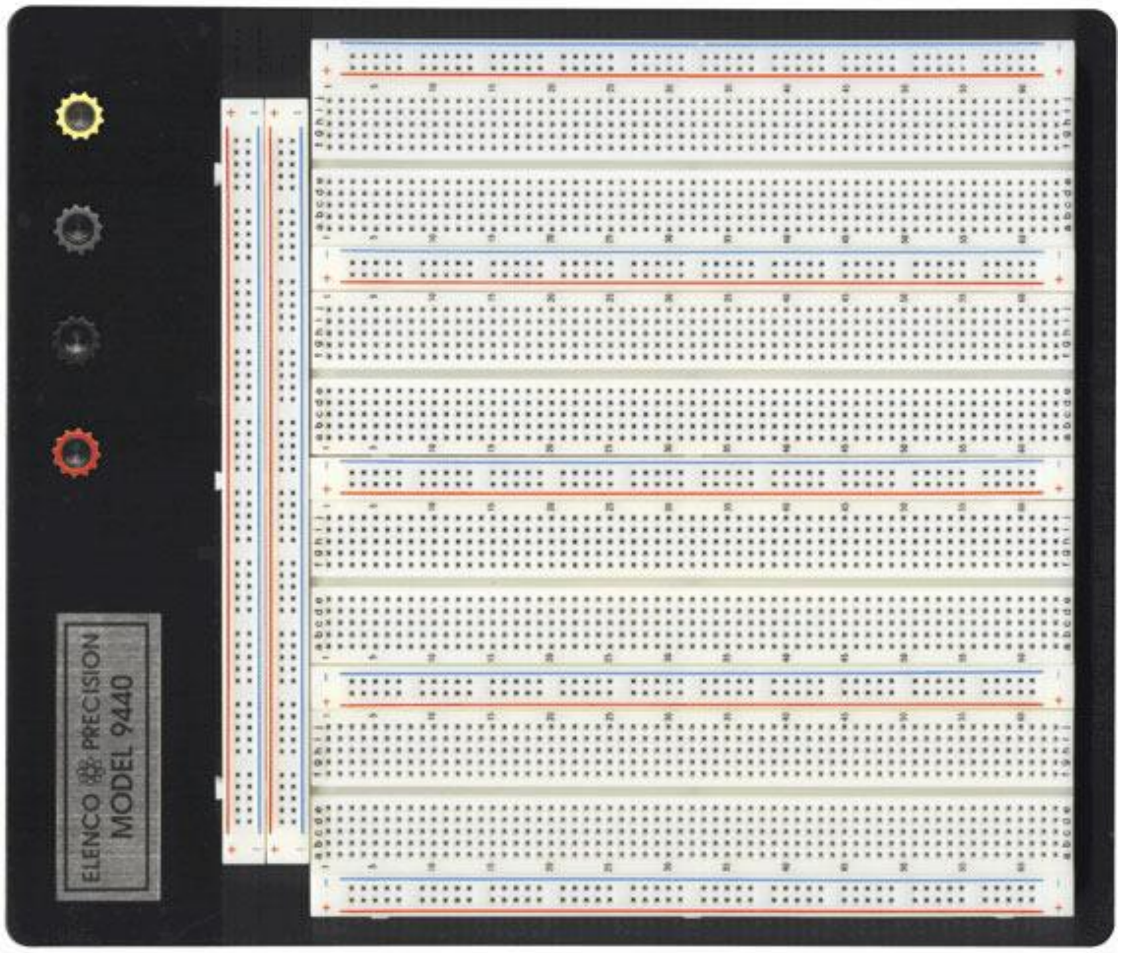
1.6.4 DER TASK: Your First Chumpanese Program

Write a program that will (hopefully) run on your CHUMP processor, six weeks from now. Your DER entry will include the entire CHUMP Instruction Set, a Table summarizing your code similar to the ones I've presented in the previous pages and a detailed explanation in works of what the program does.

2 Hardware Overview

2.1 The 4-bit TTL Processor Platform

Your ABRA-48 3220 tie-point solderless breadboard serves as sufficient prototyping real estate to complete the expected CHUMP build. Multiple supply buses across the top and separating standard breadboards serve to support power and path (address, control, data) design requirements.



2.2 Power

Detailed analysis of power requirements for functioning circuits and designing reliable sources and operating supports is a topic beyond the scope of this course and the knowledge of this instructor. Having said that, we can apply some fundamental concepts.

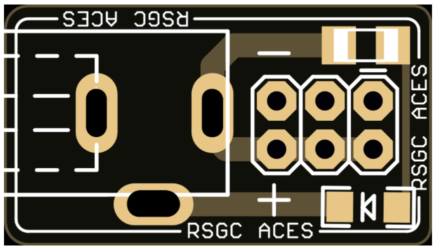
2.2.1 5V DC Voltage Source

Unlike ICs based on CMOS circuitry that can operate with a voltage supply range anywhere from 3-18V, the TTL family of ICs in our CHUMP project requires a **steady**, clean **5V** power supply (4.5-5.5V). One drawback this presents is that CHUMP does not run directly from battery power.

For this reason, the Adafruit 5V 2A switching supply pictured to the right and included in both your Grade 11 and Grade 12 kits, is an ideal *clean* source. (https://www.adafruit.com/product/276).

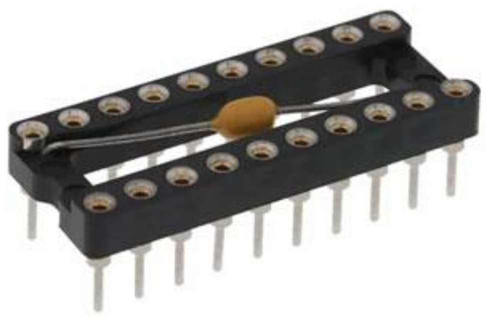
2.2.2 DC Voltage Barrel Jack Breakout Board

The RSGC ACES barrel jack breakout board, originally designed by Puneet.Bagga (*ACES '17, UofT '21*), is the ideal receptacle for your 5V power supply. In the image to the right, Puneet returned to the DES for a visit and was immediately put to work baking the SMD resistor and LED onto a dozen or so of his PCBs in the reflow oven created by Allan Hodgson (*ACES '19, Dal. '23*)



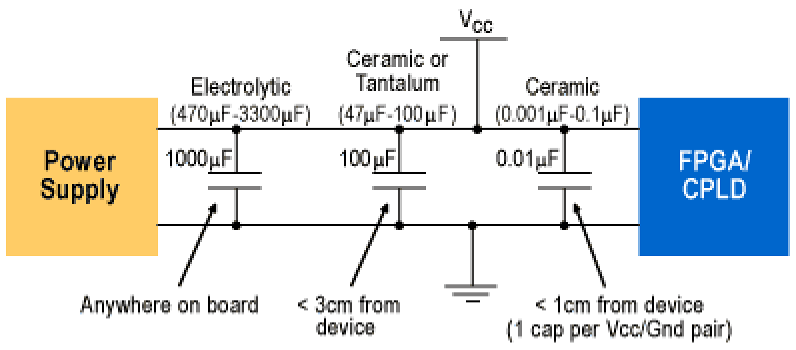
2.2.3 Bypass (Smoothing) Capacitors

Electric circuits are vulnerable to electrical noise from the proximate positioning of components, ground irregularities and voltage/current spikes that occur from rapid activation and deactivation of loads.



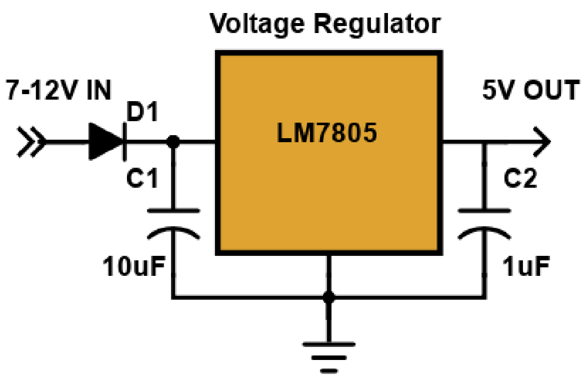
To achieve smoother power performance, strategic placement of capacitors can be beneficial. Their use is some fundamental IC sockets can be purchased (at a significant premium) with these bypass or decoupling capacitors building in.

Capacitors of various optimum sizes are used depending on the requirements, however this is beyond our knowledge and current needs. For our purposes, placing even a 10 µF capacitor across the main power rails aids in the smoothing out of supply demands to achieve *steadier* power performance. There many good sources of information on the topic which you are encouraged to review. Here's a image taken from a discussion from Intel, (https://www.intel.com/content/www/us/en/programmable/support/support-resources/operation-and-testing/power/pow-integrity.html).


Vcc
Electrolytic (470µF-3300µF)
Ceramic or Tantalum (47µF-100µF)
Ceramic (0.001µF-0.1µF)
Power Supply
1000µF
100µF
0.01µF
FPGA/ CPLD
Anywhere on board
< 3cm from device
< 1cm from device (1 cap per Vcc/Gnd pair)

2.2.4 Voltage Regulation

For the motivated ACES considering a standalone CHUMP, you may wish to consider assembling your own voltage regulation source. For this purpose, either THT or SMT 5V regulators, again, with bypass or decoupling capacitors, is recommended.



2.3 Wire Considerations

Over our past two courses you have enjoyed the relative comfort afforded by the stripped, pre-formed wire kits of breadboard-friendly lengths. However, the demanding nature of complex builds like our CHUMP requires additional considerations for the optimum wire strategy. Here is my list, roughly prioritized.

1. **Solid.** I made the mistake years ago of ordering *stranded* hookup wire instead of the solid equivalent for my breadboard prototypes. An expensive error I encourage you to avoid.

2. **Gauge**. Typical breadboard holes are designed to accept between #21and #26 AWG. I recommend #22 AWG to provide a firm fit. *Note*: Sparkfun's Resistor Kit that has been included in your previous toolkits were chosen largely because of their price. My sense is that their leads are between #26 and #28AWG making them almost too thin for breadboard use.

3. **Variable Length**. The #22 AWG jumper wire kits from Grade 10 and 11 come in stripped, pre-formed lengths ensuring single board prototyping build quality remains a priority. On a build as demanding as CHUMP, this convenience becomes prohibitively expensive. For example, Digikey sells bags of 200 pcs of each of the preformed wires you are used to for $30.00 or $0.15 per piece. ACES must do better. Although it's more effort stripping and forming your own wire lengths is far more cost-effective. This why your Grade 12 kit contains the #22 AWG, 6-colour, solid hookup wire kit pictured below, right. Included in this year' kit is a good quality wire stripper that has numerous uses beyond removing the plastic sheathing from your wires.

4. **Colour-Coded Bus Sets**. As you will soon discover, signals between the components of the processors group themselves into categories referred to as *buses*. There are buses for memory addresses, data, control, and the clock. With 6 wire colours to choose from, assigning a specific colour to a particular bus simplifies the debugging challenges and increases the overall presentation and esthetic aspects of you CHUMP build.

2.4 LEDs for Indication

The complexity of this build and confirmation of its correctness is aided substantially through the use of LEDs as signal indicators. Digital multimeters or probes are useful for testing individual pin status, but the parallel interdependencies of a processor is best captured visually.

Signals travel either individually or in groups of 4 (buses) in this project. Flat-faced, rectangular LEDs are the preferred package as multiple components can be adjacent to one another without the deflection resulting from round ones.

You have been provided with red, green and yellow varieties. Incorporate them strategically in your build.

2.5 Families of Integrated Circuits

There are a few different families of integrated circuits (TTL, CMOS) based largely on power supply characteristics. Efforts should be made to understand the naming convention employed by each family to ensure your builds incorporate compatible components.

To this point our ACES program has tended to employ CMOS-compatible technology largely due to the generous supply voltage range (3-15V). For our CHUMP build, the TTL family with its tight supply voltage range (4.75-5.25V) is preferred for its quieter noise margin (0.3V).

The 7400 series of Transistor-Transistor Logic (TTL) ICs was developed in 1964 by Texas Instruments (TI). Since other manufacturers replicated the technology, TI added the prefix **SN** to indicate their versions. These ICs are dominant in our CHUMP build.

Regardless of the manufacturer, the 7400 series of ICs should be pin-for-pin compatible. However, within the family there are differences in technologies that you need to be aware of for successful interoperability. A quick summary appears below. For further detailed information consult the Electronics Club (https://electronicsclub.info/74series.htm),Wikipedia or, preferably, the component datasheet.

IC#	Manufacturer	Rating	Vcc	Family	Details
SN54nn	Texas Instruments	Military	4.5-5.5	TTL	
SN74nn	Texas Instruments	General	4.5-5.5	TTL	
SN74LSnn	Texas Instruments	General	4.5-5.5	TTL	**L**ow-power **S**chottky
SN74HCnn	Texas Instruments	General	2.0-6.0	CMOS	**H**igh-speed **C**MOS
SN74HCTnn	Texas Instruments	General	4.5-5.5	CMOS	CMOS with TTL Compatibility
74???nn	Various	General	?	?	?

The suffix **N** on the IC# simply indicates that the component is in the (THT) PDIP package.

2.6 Summary of CHUMP Integrated Circuits

To gain an initial sense of the scope of the 4-bit CHUMP processor and for quick future reference a summary of the ICs employed is timely.

Each Sr. ACE will be assigned a particular IC with which to become intimately familiar. As questions arise, the class will defer to the respective 'expert' for additional insight and guidance.

Respective pin diagrams and details are presented later in the workbook.

IC #	Who?	Description	Usage
555	All	Timer	**Clock**: Clock pulse
SN74LS**00**	All	Quad 2-input NAND	**Branch**: Jump bit logic (Load)
SN74LS**04**	All	Hex Inverter	May require inversion of RAM output
SN74LS**08**	All	Quad 2-input AND	Optional Control Logic Support
SN74LS**32**	All	Quad 2-input OR	Optional Control Logic Support
SN74LS**157**		Quad 2/1 Data Selector (Mux)	**SEL/MUX**: Select constant and memory operand
SN74LS**161**		4-bit Counter	**PC**: Program Counter
SN74LS**174**		Hex D-Type Flip-Flop (Latch)	**Addr**: Address Register
SN74LS**181**		Arithmetic Logic Unit	**ALU**: Arithmetic and Logic Functions
SN74LS**189**		64-bit (8×8) RAM	**RAM**: Random Access Memory
SN74LS**377**		Octal D-Type Flip-Flop (Latch)	**Accum**: Accumulator Register
AT28C**16**		16K (2K×8) Parallel EEPROM	**Program**: Code
AT28C**16**		16K (2K×8) Parallel EEPROM	**Control**: Logic

3 Hardware Builds

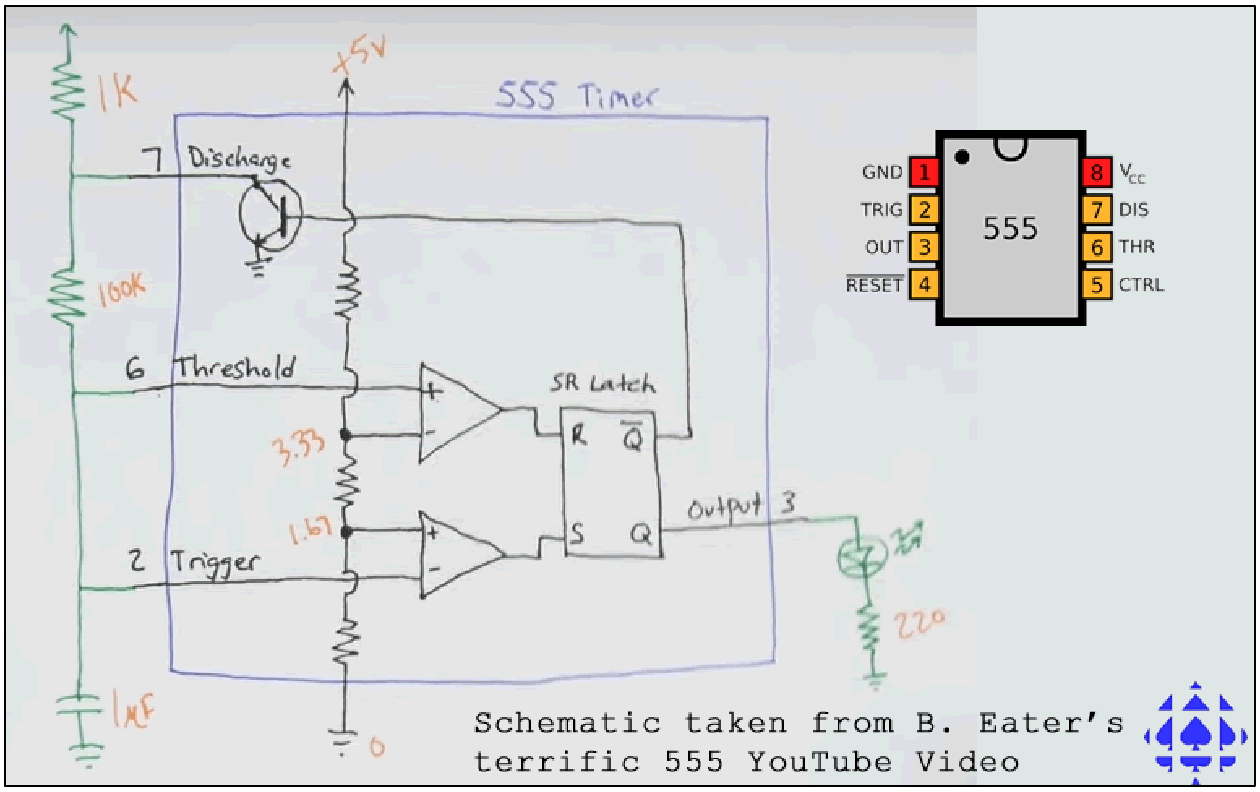
With the general overview of the hardware resources behind us, we now begin to assemble the processor.

To begin, just as the ATmega328p requires the oscillation of a crystal to define the speed and enable the synchronous coordination of the MCU assets so, too, does our processor require a heartbeat or clock of its own. Of the options available, the familiar NE555 Timer IC is a versatile choice, performing well at 5V.

3.1 555 Timer/Clock

Integrated circuits are designed to perform their various functions on the edges (*rising* or *falling*) of a square wave. Furthermore, for the multitude of ICs that make up our CHUMP processor to act sychronously, they must all be coordinated under a **single** square wave. Arguably the most versatile source for a square wave is the familiar 555 Timer IC. We'll use a number of them in our CHUMP build to provide our many ICs with a common square wave, or clock pulse.

Ben Eater is one of YouTube's most informative source of reliable electronics information. His remarkably clear set of video tutorials on the 555 clock source he developed for his own 8-bit TTL Processor (https://eater.net/8bit/clock) will form the basis of our own 4-bit, CHUMP Build.



Here are the links to Eater's 4-part computer clock video series that we will watch again.

1. Astable 555 timer - 8-bit computer clock - part 1
   https://www.youtube.com/watch?v=kRlSFm519Bo

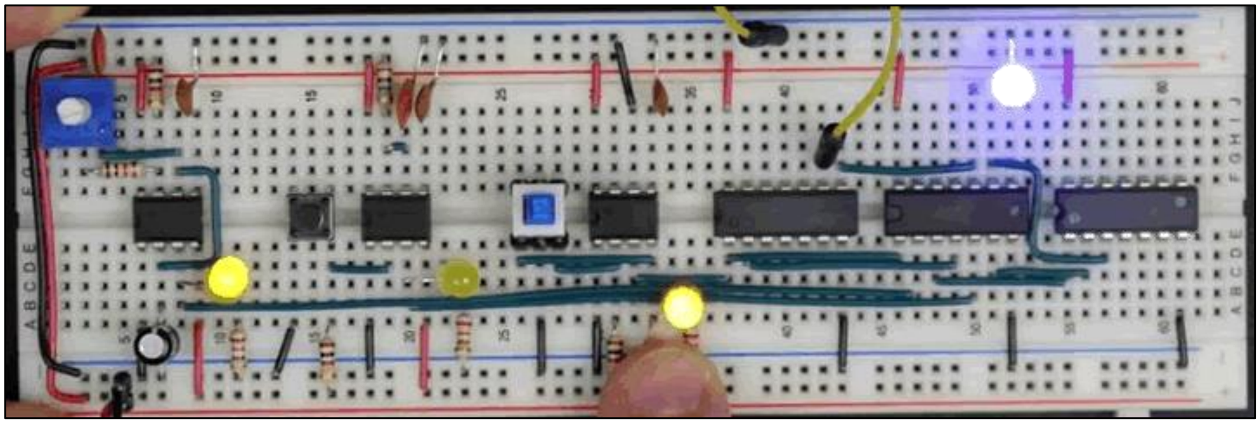
2. Monostable 555 timer - 8-bit computer clock - part 2
   https://www.youtube.com/watch?v=81BgFhm2vz8

3. Bistable 555 - 8-bit computer clock - part 3
   https://www.youtube.com/watch?v=WCwJNnx36Rk

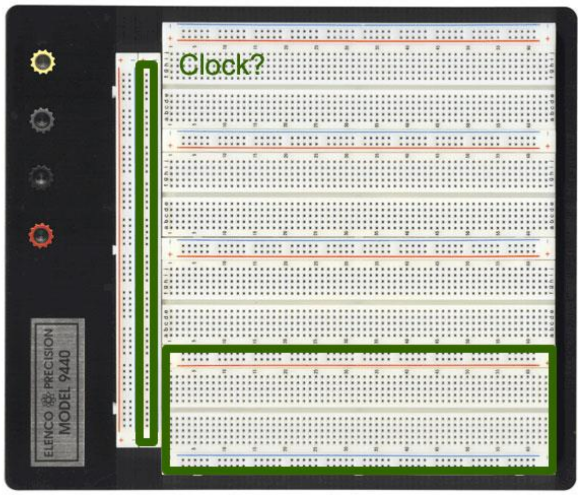
4. Clock logic - 8-bit computer clock - part 4
   https://www.youtube.com/watch?v=SmQ5K7UQPMM

3.1.1 The Clock Build

Here is an image of Eater's final clock build taken from his final video.



For our build, it is recommended to create this clock circuit on the bottom breadboard of the platform as shown to the right.



In this way, the square wave output could be placed on any of the 'vertical' supply rails to the left for access

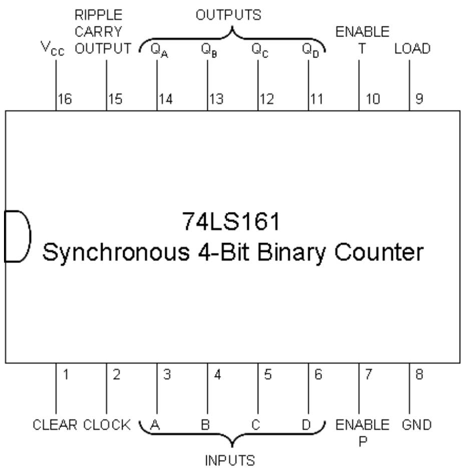
3.1.2 DER TASK: Clock Build

See ACES Task Page for the first stage of the CHUMP build.

3.2 74LS161 Synchronous Program Counter

Interestingly, this next stage is somewhat familiar to you. Our Grade 10 *Counting Circuit* employed a **NAND Gate Oscillator** as a source for its clock pulse. The square wave output from pin 10 of the CMOS 4011 IC was wired into pin 14 of the CMOS 4017 **Decade Counter** that advanced and presented its count on its 10 output pins for each *rising* edge of the NGO's pulse.

Our 5V TTL processor uses the 555 as the clock. The SN74LS161N replaces the 4017 counter from Grade 10. The 555 clock source is wired into pin 2 of the SN74LS161N. Under normal conditions the count advances, again, on the rising edge of the square wave. The output of the count appears on pins 11 through 14, as shown below.



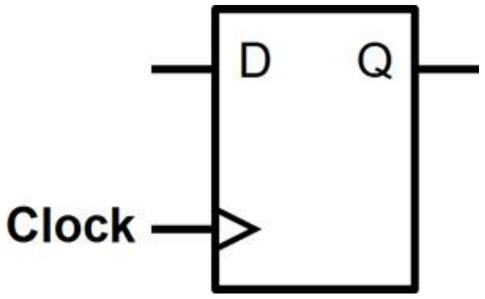
3.2.1 Program Counter Insights

- When LOAD is high, outputs will increment after each clock pulse (where Q_A is the low-order bit). When LOAD is low, outputs will instead match input data after the next clock pulse.
- CLEAR should normally be connected high. A low level at the CLEAR input will immediately set all outputs to low. (This will be useful for resetting your machine.)
- ENABLE P and ENABLE T should be wired high.

3.2.2 Clock-Influenced IC Block Diagram

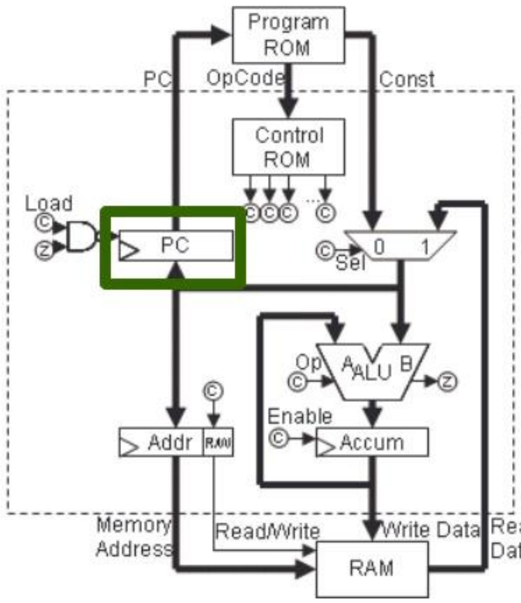
Synchronous circuit performance is achieved through the use of a common clock. The block diagram for ICs that are influenced by the clock identify the input that receives the signal by means of a small inward-facing triangle.

The block diagram of a D flip-flop that appears to the right is one such example.



3.2.3 DER TASK: Program Counter

This stage is a significant step forward as you confirm that your clock signal can influence the behavior of another IC, in a predictable manner.

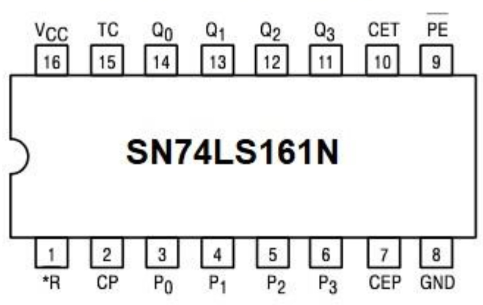


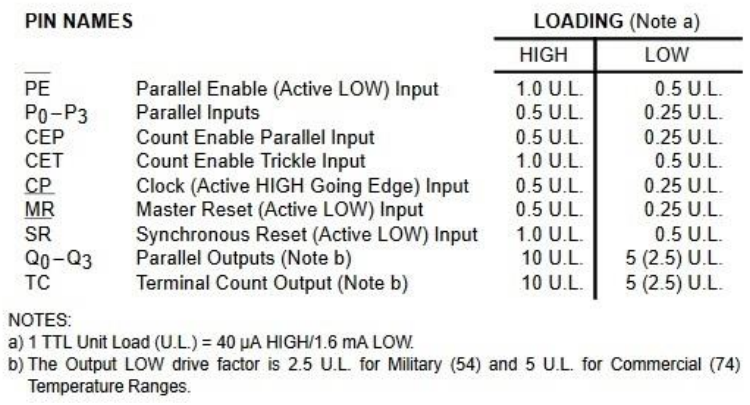
The 74LS161N IC is the BCD program counter. It is responsible for keeping track of the address within the Program EEPROM of the next instruction to be executed.

The inward-facing triangle on the program counter (PC) of the block diagram to the right indicates that it receives a clock signal. Here is an excerpt from the Motorola's datasheet on the SN74LS161N,

"*The LS160A/ 161A / 162A /163A are 4-bit synchronous counters with a synchronous Parallel Enable (Load) feature. The counters consist of four edge-triggered D flip-flops with the appropriate data routing networks feeding the D inputs. All changes of the Q outputs (except due to the asynchronous Master Reset in the LS160A and LS161A) occur as a result of, and synchronous with, the LOW to HIGH transition of the clock input (CP).*"

CONNECTION DIAGRAM




PIN NAMES

		LOADING (Note a)	
		HIGH	LOW
PE	Parallel Enable (Active LOW) Input	1.0 U.L.	0.5 U.L.
P_0-P_3	Parallel Inputs	0.5 U.L.	0.25 U.L.
CEP	Count Enable Parallel Input	0.5 U.L.	0.25 U.L.
CET	Count Enable Trickle Input	1.0 U.L.	0.5 U.L.
CP	Clock (Active HIGH Going Edge) Input	0.5 U.L.	0.25 U.L.
MR	Master Reset (Active LOW) Input	0.5 U.L.	0.25 U.L.
SR	Synchronous Reset (Active LOW) Input	1.0 U.L.	0.5 U.L.
Q_0-Q_3	Parallel Outputs (Note b)	10 U.L.	5 (2.5) U.L.
TC	Terminal Count Output (Note b)	10 U.L.	5 (2.5) U.L.

NOTES:
a) 1 TTL Unit Load (U.L.) = 40 µA HIGH/1.6 mA LOW.
b) The Output LOW drive factor is 2.5 U.L. for Military (54) and 5 U.L. for Commercial (74) Temperature Ranges.

3.2.3.1 Placement of the Program Counter

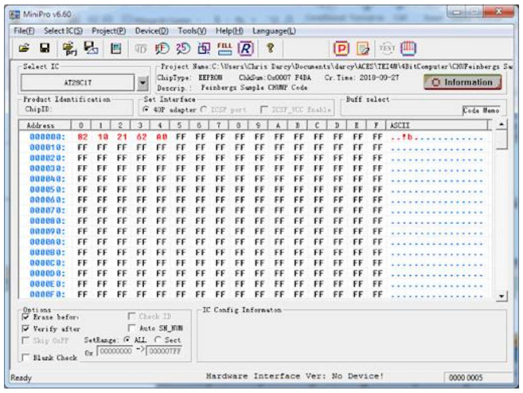
Since the majority of components of the processor are still ahead of us there is little need to be precise about its placement on the breadboard and the specific wire characteristics to tie it in. Nevertheless, I include recommendations from Feinberg on this stage,

"*Note that the counter and ROM chips we'll be using in this lab will have a permanent home on our boards. It is recommended that you place the counter in the upper left corner of your board, and eventually place the ROM immediately to the right of the counter. Wires to switches and LEDs will be temporary, but other wires will be permanent features of your board, and therefore should be wired very neatly.*"

3.3 AT28C16 Parallel EEPROM

Unlike volatile RAM (Random Access Memory) that loses its contents when the power supply is removed, EEPROM (Electronically Erasable Programmable Read Only Memory) is a form of non-volatile storage that retains its contents after the power is disconnected. The microprocessors prior to the 1980s relied on the use of these external EEPROMs. Microcontrollers eventually evolved as devices with an embedded microprocessor, EEPROM, ADC and DAC units, Timers, Counters and other peripherals. Through your familiarity with the ATMEL MCUs in Grade 11, you have become accustomed to exploiting the non-volatile ability of the builtin EEPROM

Our CHUMP requires **two** parallel EEPROM ICs: one for your **program** code and one for instruction **control** codes (*used in lieu of combinational logic*). You have been supplied with two AT28C16 ICs. The earlier ATMEL EEPROM ICs (AT28C16, AT28C17, etc.) as called for in Feinberg's original design, are becoming increasingly harder to source for this build. Fortunately, the EEPROM ICs in this family, with larger memory capacity (ie. AT28C256) are compatible and available, but are expensive.


MiniPro v6.60
File(F) Select IC(S) Project(P) Device(D) Tools(V) Help(H) Language(L)
Information

```
Select IC: AT28C17
Project Name:C:\Users\Chris Darcy\Documents\darcy\ACES\TEI4M\4BitComputer\CHUFeinbergs Sa
ChipType: EEPROM    ChkSum:0x0007 F4BA    Cr.Time: 2018-09-27
Descrip.: Feinbergs Sample CHUMP Code
Product Identification  ChipID:
Set Interface: (•) 40P adapter  ( ) ICSP port  [ ] ICSP_VCC Enable
Buff select: Code Memo

Address   0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F  ASCII
000000:  82 10 21 62 A0 FF FF FF FF FF FF FF FF FF FF FF  ..!b............
000010:  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
000020:  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
000030:  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
000040:  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
000050:  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
000060:  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
000070:  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
000080:  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
000090:  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
0000A0:  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
0000B0:  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
0000C0:  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
0000D0:  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
0000E0:  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
0000F0:  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................

Options
[✓] Erase before     [ ] Check ID
[✓] Verify after     [ ] Auto SN_NUM
[ ] Skip 0xFF        SetRange: (•) ALL  ( ) Sect
[ ] Blank Check      0x 00000000 -> 000007FF
IC Config Informaton

Ready          Hardware Interface Ver: No Device!          0000 0005
```

Changes to your Arduino code are as simple as pressing the upload button in the IDE to have the AVRDUDE program manage the communication protocol through your USB cable. With our standalone CHUMP processor, the task of flashing the code we wish to execute is not quite as straightforward, but we do have three alternatives to accomplish thetask as described in the next three sections.

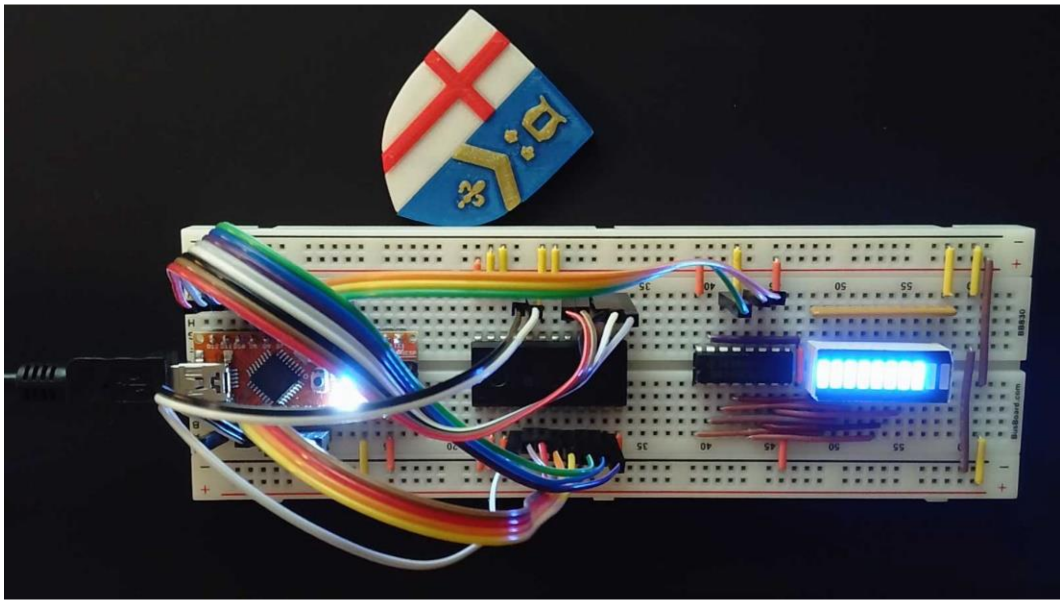
A CHUMP **program** consists of a *maximum* of 16 instructions. This limitation is a direct consequence of the 4-bit address bus. The EEPROM address of a program's instruction is that instruction's **line number**. Line numbers can range from `0000` to `1111`. Line numbers also serve as target addresses for the two branch instructions.

3.3.1 TL866 II EEPROM Programmer

One option for flashing the contents of your 8-bit parallel EEPROM IC is the use of a hardware programmer. The TL866 II EEPROM Programmer is one such example that ACES have used in the past and is available from Eater's shop or Amazon to name two sources. As a Windows-only, software-driven device and at a cost of about $100, this may not be the best option for ACES.

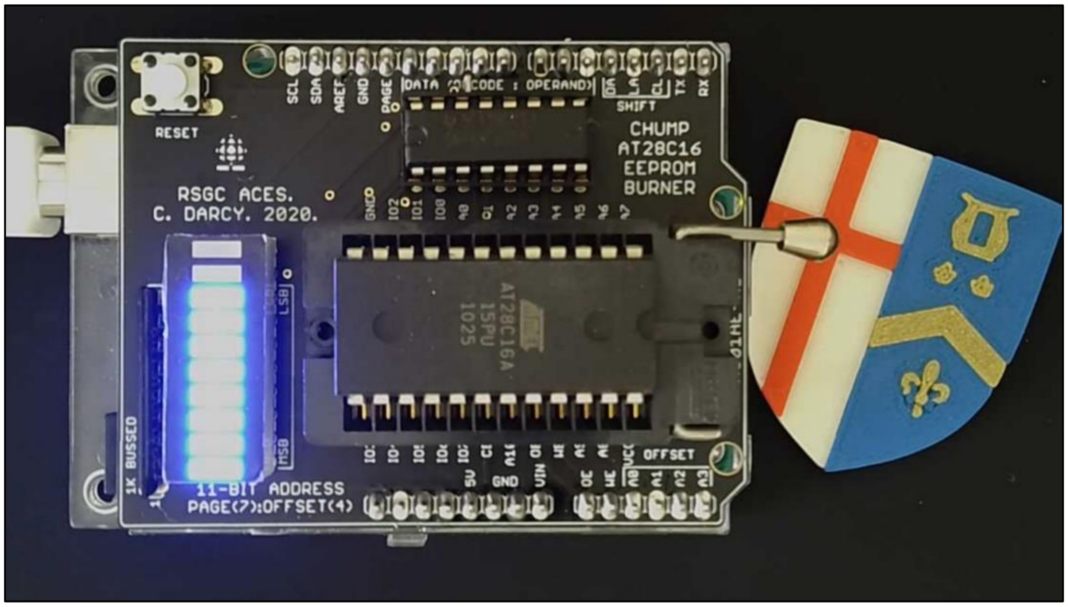
3.3.2 Breadboard EEPROM Programmer

The least expensive and readily-available EEPROM programming option is to place your EEPROM on a breadboard wired to your Nano (or UNO) as shown below. The prototype below includes an optional shift register/bargraph combination for read and display confirmation. Code will be provided in class. Here's a short video of the device flashing the `SwappingVariables` example: https://tinyurl.com/w24vn6sv.



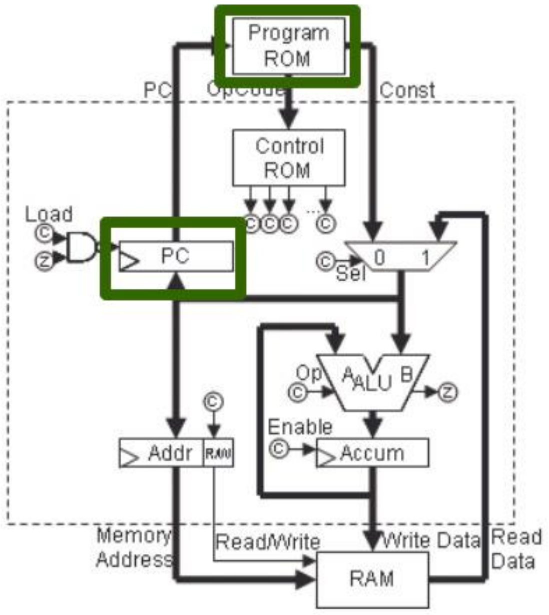
3.3.3 RSGC ACES AT28C16 EEPROM Programmer Shield

The third option is perhaps the cleanest. In January 2020 a custom AT28C16 EEPROM Programmer Shield was developed for the UNO. An onboard 24-pin ZIF Socket allows for rapid insertion and removal of its delicate pins. The onboard shift register/bargraph combination provides read and display confirmation. Code will be provided in class. Here's a short video of the device flashing Feinberg's Sample code: https://tinyurl.com/vz36u76u.



3.3.3 DER TASK: Program EEPROM

This stage is the first opportunity to combine your hardware with the software you wrote in the first stage.

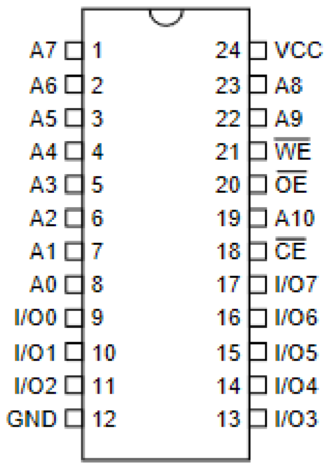


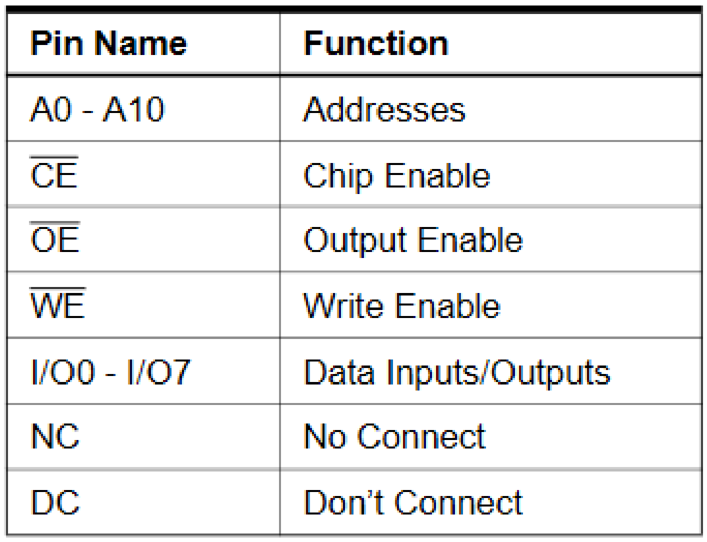
Step 1. Once you have obtained the machine language version of your code, enter it into the MiniPro application and flash it into your AT28C16 EEPROM IC.

Step 2. Place your Program EEPROM in close proximity with your Program Counter on the breadboard. The BCD outputs of the PC are wired into the A0-A3 address lines of the Program EEPROM. The remaining 7 address lines (A4-A10) must be grounded (for now:).

Step 3. Once the ICs are wired in, rectangular LEDs on the 8 output pins of the AT28C16 will (should) confirm your CHUMPanese machine code.

3.3.3.1 AT28C16 16K (2K×8) Parallel EEPROM Pin Connections





Pin Name	Function
A0 - A10	Addresses
$\overline{CE}$	Chip Enable
$\overline{OE}$	Output Enable
$\overline{WE}$	Write Enable
I/O0 - I/O7	Data Inputs/Outputs
NC	No Connect
DC	Don't Connect

When CE and OE are low and WE is high, the data stored at the memory location determined by the address pins is asserted on the outputs.

3.3.4 Program EEPROM Pagination

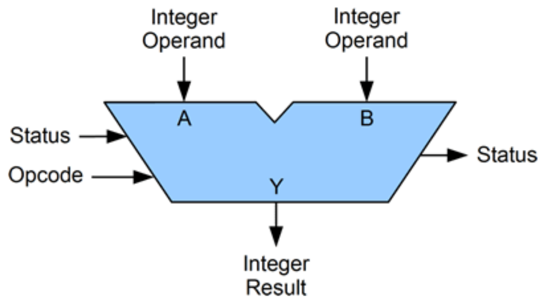
Even the AT28C17 EEPROM has far more memory than the 4-bit address bus of the CHUMP can access. This presents an interesting challenge for ACES to find a way to flash and access multiple programs!

- AT28C17 has 2K×8 bits of storage
- 2K is 2^{11} so each byte address requires 11 bits
- CHUMP Program is limited to 16×8 bits
- Result: room for 2^7 or 128 separate CHUMP programs
- Divide EEPROM up into 128 'pages' with each page holding a CHUMP program
- Addresses are of the form PAGE:OFFSET and can be thought of as **PPP** PPPP OOOO
- RSGC ACES EEPROM Burner Shield uses A0 through A4 or 5 address bits (32 programs)
- Restricting PAGE to 1 set bit offers access to 8 different pages (shaded blue, below)

	000		001		010		011		100		101		110
0000	A		F		G				H				
0001	B												
0010	C												
0011													
0100	D												
0101													
0110													
0111													
1000	E												
1001													
1010													
1011													
1100													
1101													
1110													
1111													

3.4 74LS181 Arithmetic Logic Unit (ALU)

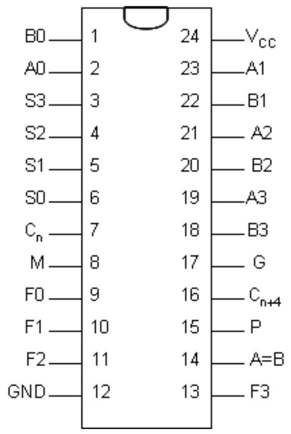
In September 1979 I started teaching my first course in Computer Science at RSGC. It wasn't until the start of the second term, in January 1980, that we took delivery of some hardware on which students were able to run their code in our classroom. The computer we purchased for $25,000 was a DEC PDP-11/03 minicomputer, pictured to the right. A year later our 'Ladies Guild' purchased three VT100 terminals that allowed students to enter and run their programs allowing us to dispense with the card reader. The interesting feature of this rather expensive hardware was that its CPU was based on the same SN74LS181 4-bit Arithmetic Logic Unit you have in front of you that sells for under $2.00. The Arithmetic Logic Unit (ALU) of a CPU is the IC containing circuitry to perform single or double operand arithmetic or logical functions. The circuit symbol for and ALU is as follows,



3.4.1 The 74LS181 ALU

Ken Shirriff, of previous IR and PWM fame, devotes a wonderful blog to explaining the mysteries of the 74181 ALU that even includes an interactive simulation of the ALU,

http://www.righto.com/2017/03/inside-vintage-74181-alu-chip-how-it.html



Pin Names	Description
A0 – A3	Operand Inputs
B0 – B3	Operand Inputs
S0 – S3	Function Select Inputs
M	Mode Control Input
C_n	Carry Input
F0 – F3	Function Outputs
A=B	Comparator Output
G	Carry Generate Output
P	Carry Propagate Output
C_{n+4}	Carry Output

The A=B output from the device goes high when all four F outputs are high. The A=B output is open-collector, meaning that it should be connected via a 2.2KΩ resistor to +5 volts.

3.4.1.1 74LS181 ALU Function Table

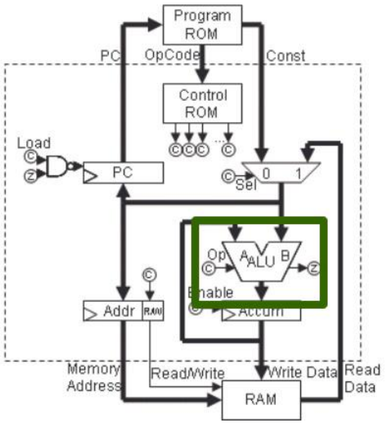
S3	**S2**	**S1**	**S0**	**Logic (M = H)**	**Arithmetic (M = L) (C_n = H)**
L	L	L	L	¬A	A
L	L	L	H	¬ (A or B)	A or B
L	L	H	L	¬A and B	A or ¬B
L	L	H	H	Logic 0	minus 1
L	H	L	L	¬(A and B)	A plus (A and ¬B)
L	H	L	H	¬B	(A or B) plus (A and ¬B)
L	H	H	L	A xor B	A minus B minus 1
L	H	H	H	A and ¬B	(A and B) minus 1
H	L	L	L	¬A or B	A plus (A and B)
H	L	L	H	¬(A xor B)	A plus B
H	L	H	L	B	(A or ¬B) plus (A and B)
H	L	H	H	A and B	(A and B) minus 1
H	H	L	L	Logic 1	A plus A
H	H	L	H	A or ¬B	(A or B) plus A
H	H	H	L	A or B	(A or ¬B) plus A
H	H	H	H	A	A minus 1

- Arithmetic operations expressed in 2s complement notation.
- In arithmetic mode (M = L), setting C_n = L adds 1 to output.

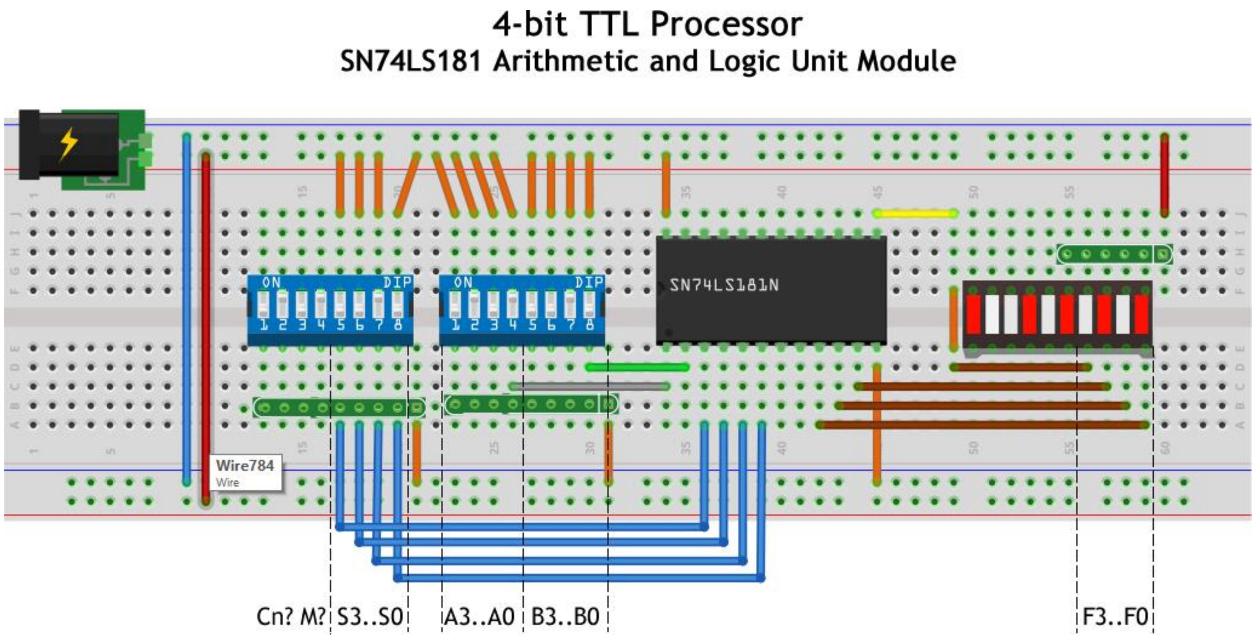
3.4.2 DER TASK: The ALU

In this segment of your CHUMP journey you are asked to confirm your functional knowledge of the SN74LS181 ALU in isolation, prior to integrating it into your full processor build. A summary of the ALU's function table appears below, left.

SELECTION				ACTIVE-HIGH DATA		
				M = H	M = L; ARITHMETIC OPERATIONS	
S3	S2	S1	S0	LOGIC FUNCTIONS	$\overline{C_n}$ = H (no carry)	$\overline{C_n}$ = L (with carry)
L	L	L	L	F = $\overline{A}$	F = A	F = A PLUS 1
L	L	L	H	F = $\overline{A + B}$	F = A + B	F = (A + B) PLUS 1
L	L	H	L	F = $\overline{A}B$	F = A + $\overline{B}$	F = (A + $\overline{B}$) PLUS 1
L	L	H	H	F = 0	F = MINUS 1 (2's COMPL)	F = ZERO
L	H	L	L	F = $\overline{AB}$	F = A PLUS A$\overline{B}$	F = A PLUS A$\overline{B}$ PLUS 1
L	H	L	H	F = $\overline{B}$	F = (A + B) PLUS A$\overline{B}$	F = (A + B) PLUS A$\overline{B}$ PLUS 1
L	H	H	L	F = A ⊕ B	F = A MINUS B MINUS 1	F = A MINUS B
L	H	H	H	F = A$\overline{B}$	F = A$\overline{B}$ MINUS 1	F = A$\overline{B}$
H	L	L	L	F = $\overline{A}$ + B	F = A PLUS AB	F = A PLUS AB PLUS 1
H	L	L	H	F = $\overline{A \oplus B}$	F = A PLUS B	F = A PLUS B PLUS 1
H	L	H	L	F = B	F = (A + $\overline{B}$) PLUS AB	F = (A + $\overline{B}$) PLUS AB PLUS 1
H	L	H	H	F = AB	F = AB MINUS 1	F = AB
H	H	L	L	F = 1	F = A PLUS A	F = A PLUS A PLUS 1
H	H	L	H	F = A + $\overline{B}$	F = (A + B) PLUS A	F = (A + B) PLUS A PLUS 1
H	H	H	L	F = A + B	F = (A + $\overline{B}$) PLUS A	F = (A + $\overline{B}$) PLUS A PLUS 1
H	H	H	H	F = A	F = A MINUS 1	F = A



Your parts kit includes a pair of DIP-8 rocker switches that are to be laid out as shown below, suitably pulled down through the use of 1kΩ resistor networks. Rectangular LEDs are suitable replacements for the bargraph.

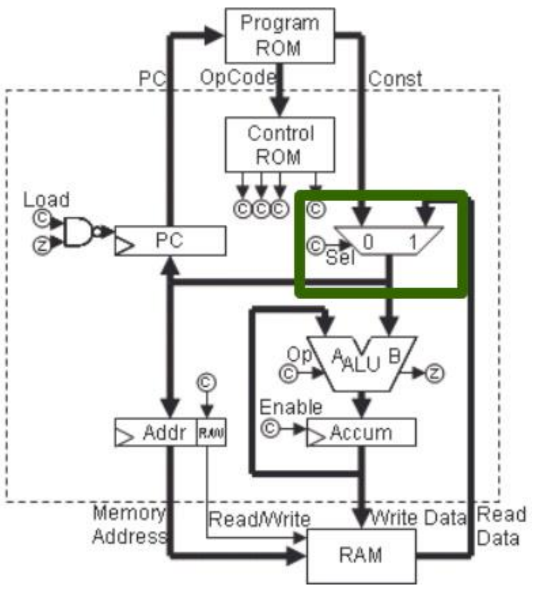


Your video should include both a discussion of your build as well as a sampling of a few arithmetic and logic functions.

3.5 74LS157 Multiplexer (Selector)

Since the operand of a CHUMP instruction can be either immediate (constant) or a RAM Address, our processor needs a way to select between the two.

A multiplexer (mux) is an IC consisting of internal combinational logic that performs this selection task. The circuit symbol for a mux appears highlighted in the adjacent CHUMP block diagram.

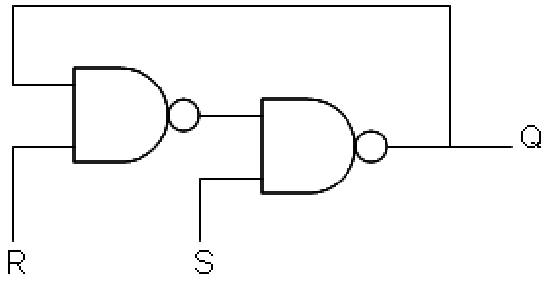


Inputs known as select pins are employed by the IC to determine which of the input busses should be passed through to the output bus. For CHUMP purposes, since there are only **two** input buses (Const and Read Data), each four bits wide, only one select pin is required to choose between the two. The 74LS157 Quad 2-Line to 1-Line Data Selector/Multiplexer fulfills this requirement.

3.6 Memory

3.6.1 Latch

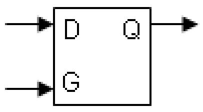
Combinational circuitry (that forms in the internal logic of the 74LS157 mux IC) is such that the output depends solely on the inputs at any given instant.



Sequential circuitry, on the other hand, combines the current inputs from the previous state to define the outputs. For example, two NAND gates wired as shown to the right, employs feedback to yield the basis for an SR Latch. Latches are one type of building block of computer memory.

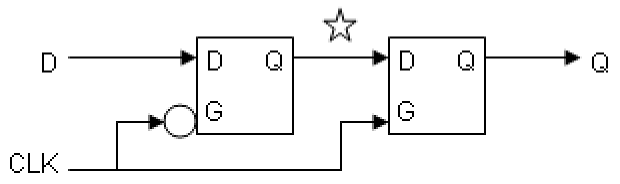
3.6.2 D-Type Flip-Flop

One can build on the concept of a latch to create a device represented by the adjacent symbol. When G (the "Gate") is 1 (open), the value of the output Q should match the value of D. When G is 0 (closed), Q should continue to output whatever value it had when the gate closed. This behavior is summarized in the following table.

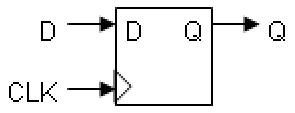


G	D	Q_{OLD}	Q
0	X	0	**0**
0	X	1	**1**
1	0	X	**0**
1	1	X	**1**

In theory, we can connect two latches to create a flip-flop as follows.



This device is called a flip-flop, and acts like a tollbooth with two gates. When one gate is up, the other is down. This way only one value can get through at once. The CLK ("clock") signal alternates between low and high values, so as to change which gate is up. When the CLK signal goes up, the original input D has made it all the way to output Q. We say that a flip-flop is "edge-triggered", and that its output changes on the "rising clock edge". We'll use the following symbol to represent flip-flops.

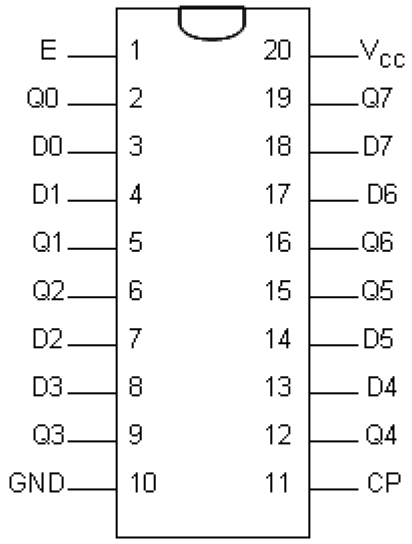


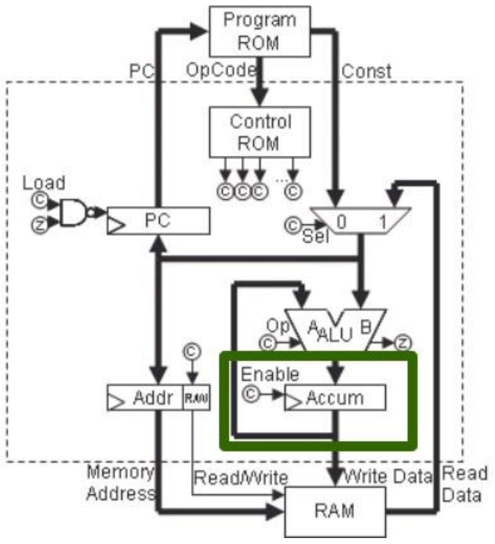
Unfortunately, the actual construction of a flip-flop relies on very delicately timed gates. We will consider such timing issues to be beyond the scope of this course, so we will therefore not be constructing our own flip-flops.

3.6.3 74LS377 Accumulator (Octal Quad D-Type Flip-Flops)

E	CP	D_n	Q_n
H	X	X	**No Change**
L	↑	H	**H**
L	↑	L	**L**

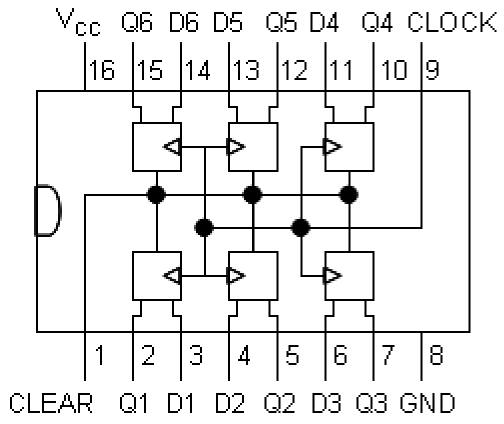
Upon completion of a requested arithmetic or logic function, the ALU must have somewhere to store the result for future use. From the CHUMP block diagram, it is evident that it is the Accumulator that is the direct recipient of the output of the ALU. Furthermore, a datapath from the Accumulator to the A input of the ALU is also required. The clocked 74LS377 Octal Quad D-Type Flip-Flop IC serves to support the role as the immediate storage register in support of the ALU's output.

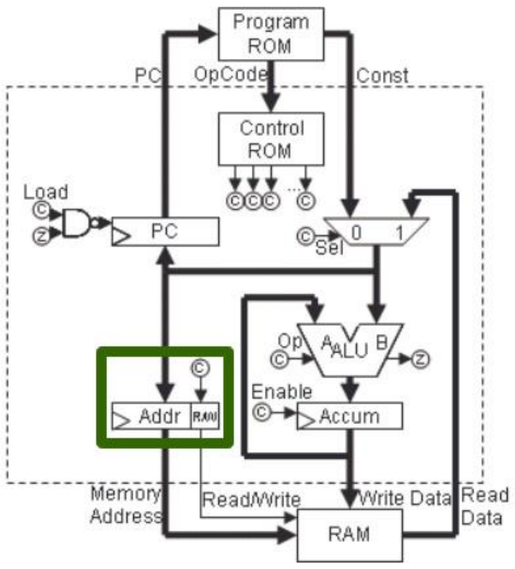




3.6.4 74LS174 Address (Hex D-Type Flip-Flop)

The contents of the Accumulator typically change with every ALU function. For your code to be useful, the contents of Accumulator will require more permanent residence in Random Access Memory (RAM). RAM is similar to a bank of bits (organized in nibbles of bytes), similar to a set of mailboxes, each with its own unique address. Access to a particular storage location in the CHUMP's RAM IC (74LS289) requires an Address IC, the 74LS174. Since our address bus is 4 bits wide, Feinberg employed a 74LS174 Hex D-Type Flip-Flop IC, shown in pin connection diagram below, left.





The interdependency between the pins is summarized in the function table (for each flip-flop) below. In other words, the CLEAR pin must be wired HIGH to enable the flip-flops.

Clear	Clock	D	Q
L	X	X	**L**
H	↑	H	**H**
H	↑	L	**L**
H	L	X	**Q_0**

H = HIGH LEVEL (steady state)

L = LOW LEVEL (steady state)

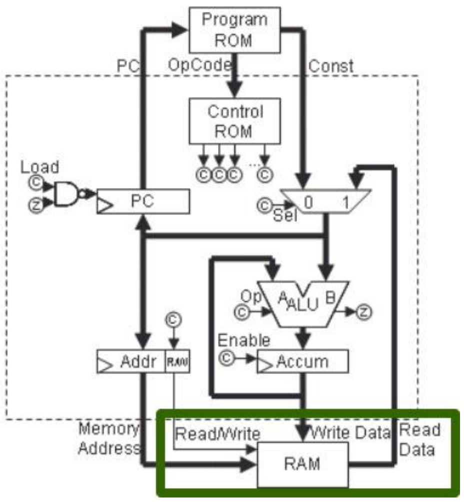
X = Don't Care

↑ = Transition from LOW-to-HIGH level

Q_0 = The level of Q before the indicated steady-state input conditions were established

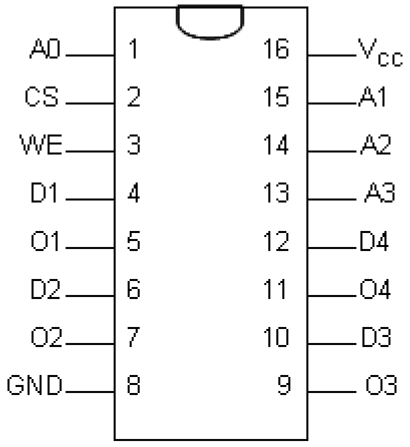
3.6.5 74LS189/289 RAM (64-bit Random Access Memory)

The 74LS189/289 RAM IC used in our CHUMP processor is the perfect fit for the sizes of our address (16) and data (4) paths, combining to yield its 64-bit capacity.



Since the four output pins are open collector types (hence, pull up resistors required), the output data is the complement of the stored data. If we wish to use the output data as is, we either need to invert data before storing it, or afterwards through the use of the 74LS04 hex inverter.

From the Instruction Table of Section 1, `mem[]` operations require some programming care. `READ`, `LOAD` and `STORETO` instructions allow data to be read from (2 cycles) or written to (1 cycle).



Pin Names	Description
A0 – A3	Address Inputs
CS	Chip Select Input (Active LOW)
WE	Write Enable Input (Active LOW)
D1 – D4	Data Inputs
O1 – O4	Inverted Data Outputs (Open Collector)

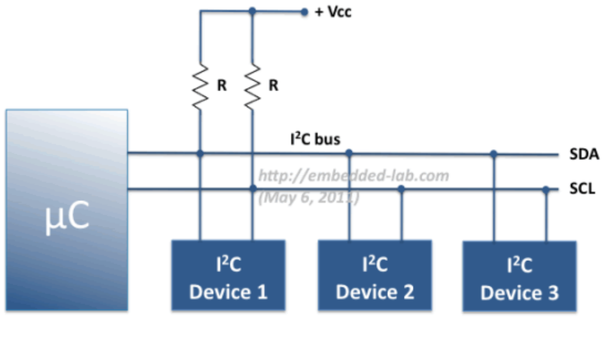
CS	WE	Operation	Condition of Outputs
L	L	Write	Off
L	H	Read	Complement of Stored Data
H	X	Off	Off

4 CHUMP Glue: The Buses

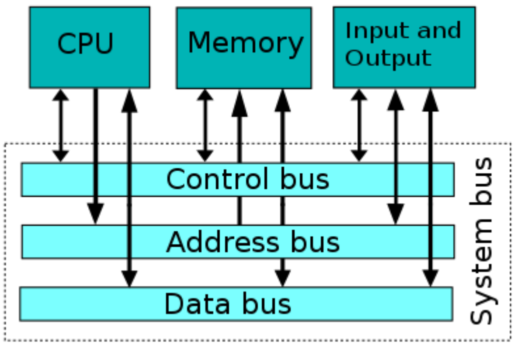
With the majority of CHUMP processor blocks introduced, some of which independently implemented, it is time to discuss the communication strategies that allow for the synchronous and communicative interdependency of the many components.

The *synchronous* ability of a computer is handled by device connections to a common clock source; in our case, the 555 Timer. The responsibility for the shared communication between blocks rests with the concept of a bus.

Communication is achieved through the use of a *bus*; which is little more than a common set of wires, each wire carrying a *bit* (0 or 1) of information. The idea of a bus is not new to ACES. In Grade 11 you were introduced to a 2-wire (SDA, SCL) method of shared communication between your ATmega328p MCU and (up to) 127 devices known as the I^{2}C bus (below, left).



Multiple devices on common I²C bus



The Wikipedia image (above, right) is an abstract generalization of a computer's (collective) *System bus*. The System bus permits the major blocks to access three separate sub-buses (Control, Address, and Data). Our CHUMP exploits these three buses as well. Our Address and Data buses are 4-wire (bit) buses, whereas our Control bus consists of 8 (or 9) wires.

4.1 The Address Bus

Within a modern computer the concepts of *address* and *data* are easily blurred. On the other hand, for our CHUMP processor, they are more easily separated. Let's look at examples of some specific components that make use of addresses (not the complete list).

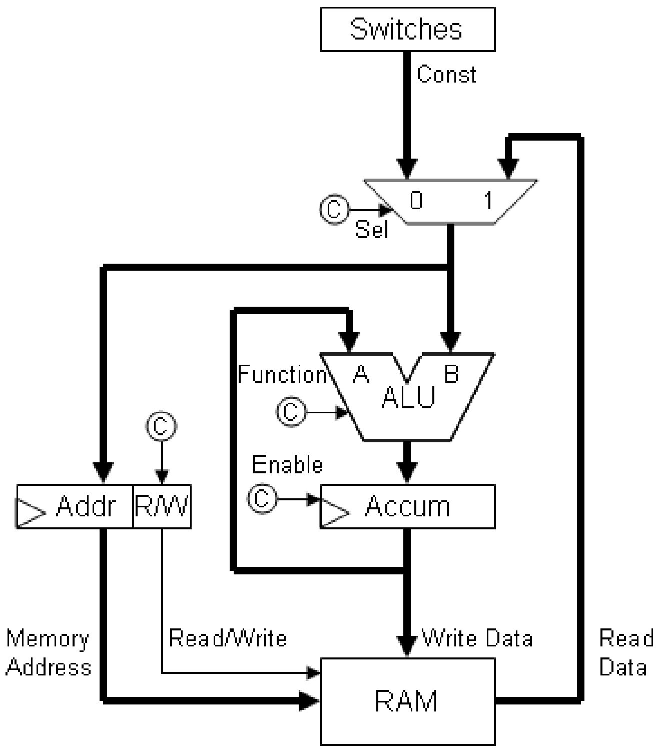
The AT28C16/17 Program EEPROM holds our CHUMPanese instructions in consecutive memory locations identified by a 4-bit address, beginning at 0000_2, to a maximum address of 1111_2.

The 74LS161 Program Counter has the responsibility for maintaining the address of the next instruction in the Program EEPROM to be executed. It is advanced by 1 on the rising edge of the clock signal or assigned a specific 4-bit value in response to the code's execution of either a GOTO or IFZERO instruction.

The 74LS189/289 Random Access Memory uses 4-bit addresses (0000_2 - 1111_2) to store 4-bit data values typically received as output from the Accumulator or input to the B-operand of the ALU.

4.2 The Data Bus

For this discussion, we'll interpret *data* to be the *operands* of ALU functions. As such, they are sourced either as lower-nibble constant of immediate-mode CHUMP instructions or values obtained by reading a RAM location. The choice between the two is controlled by means of the 74LS157 2-to-1 Multiplexer/Selector.



Feinberg's original notes encourages students to implement a partial CHUMP build as shown in the block diagram to the right. Here were his instructions to his students, "*Go ahead and build the following datapath. Wire the accumulator's outputs to 4 LEDs. Temporarily wire 4 switches to the 0 input of the selector. For each of the 9 control inputs, use a loop of wire to hard-wire the input to ground or +5 volts. Finally, wire the output of an RS circuit to the clock inputs, so that you can manually simulate running one instruction at a time.*

You will also need to fill in the following control table, and use it to test your CHUMP."

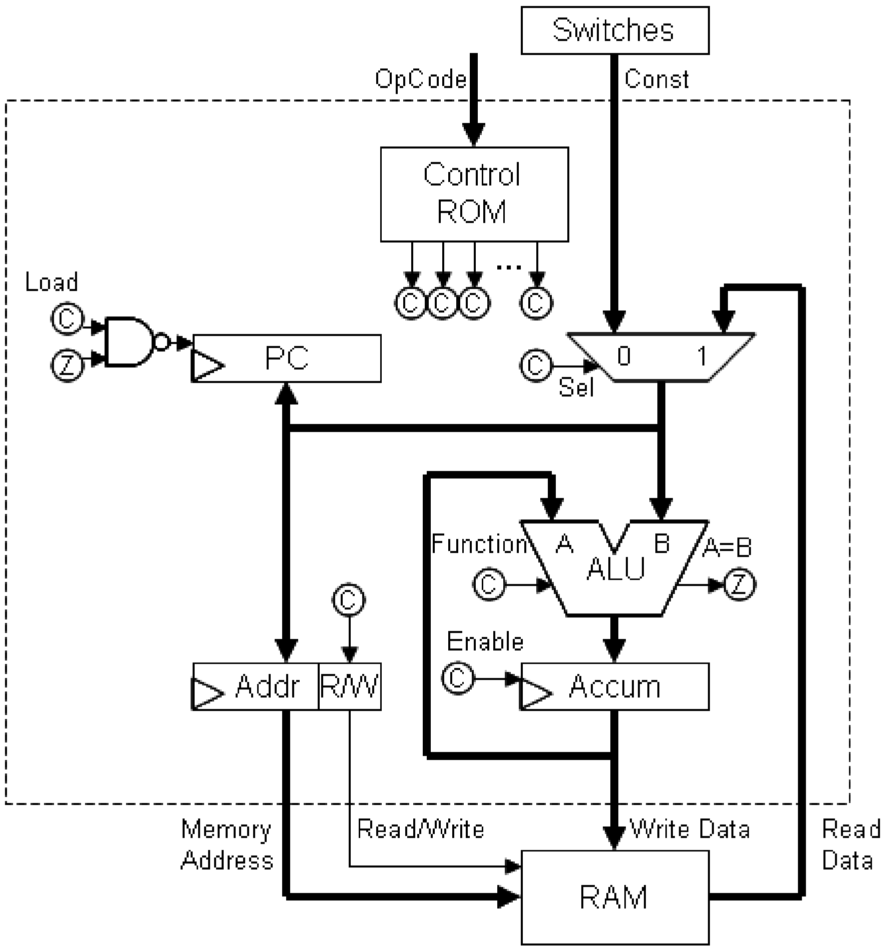
Instruction	Sel	(ALU)	S3	S2	S1	S0	M	Cn	Accum	R/W
`LOAD const`	0	B	1	0	1	0	1	X	0	1
`LOAD IT`										
`ADD const`										
`ADD IT`										
`SUBTRACT const`										
`SUBTRACT IT`										
`STORETO const`										
`STORETO IT`										
`READ const`										
`READ IT`										

4.3 The Control Bus

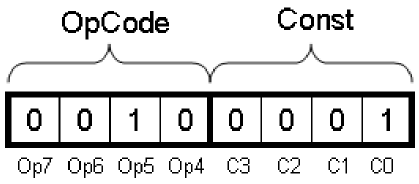
Little information has been included in this workbook to this point on the control codes indicted by the symbol, **©**, that appear as inputs to various components within the block diagram. Suffice-it-to-say, collectively they form the coordinated *control* bus. The control bus is akin to the orchestra leader that instructs which instruments enter and depart the symphony throughout the playing of the piece.

Feinberg made mention of the partial set of control codes in the previous section. In this section, we flesh out the full set of control codes. Once fully identified, these codes will be flashed into the second AT28C17 EEPROM in a manner similar to the flashing of the Program EEPROM. The Control EEPROM serves as a replacement for otherwise complicated combinational circuitry. I will continue with Feinberg's original instructions to his students,

"*In this part, we will make two improvements. First, we'll add a program counter to our processor, but we won't use its output yet. Wire the A=B output of the ALU to a NAND, and connect this NAND to the LOAD input of the program counter. This will enable us to use a control bit and the A=B output to decide whether to increment the program counter or to load a new value into the program counter instead.*"



"*Eventually, we'll use the counter's output as the address into the Program ROM where the current instruction can be found. Each instruction will consist of an 8-bit value, where the 4 high-order bits comprise the OpCode and the 4 low-order bits comprise the constant value. For example, the OpCode for the ADD-const instruction is 2. Thus, the instruction "ADD 1", which adds 1 to the accumulator, is represented by the 8-bit value 00100001, shown here. For convenience, each of these bits has been given a name.*"



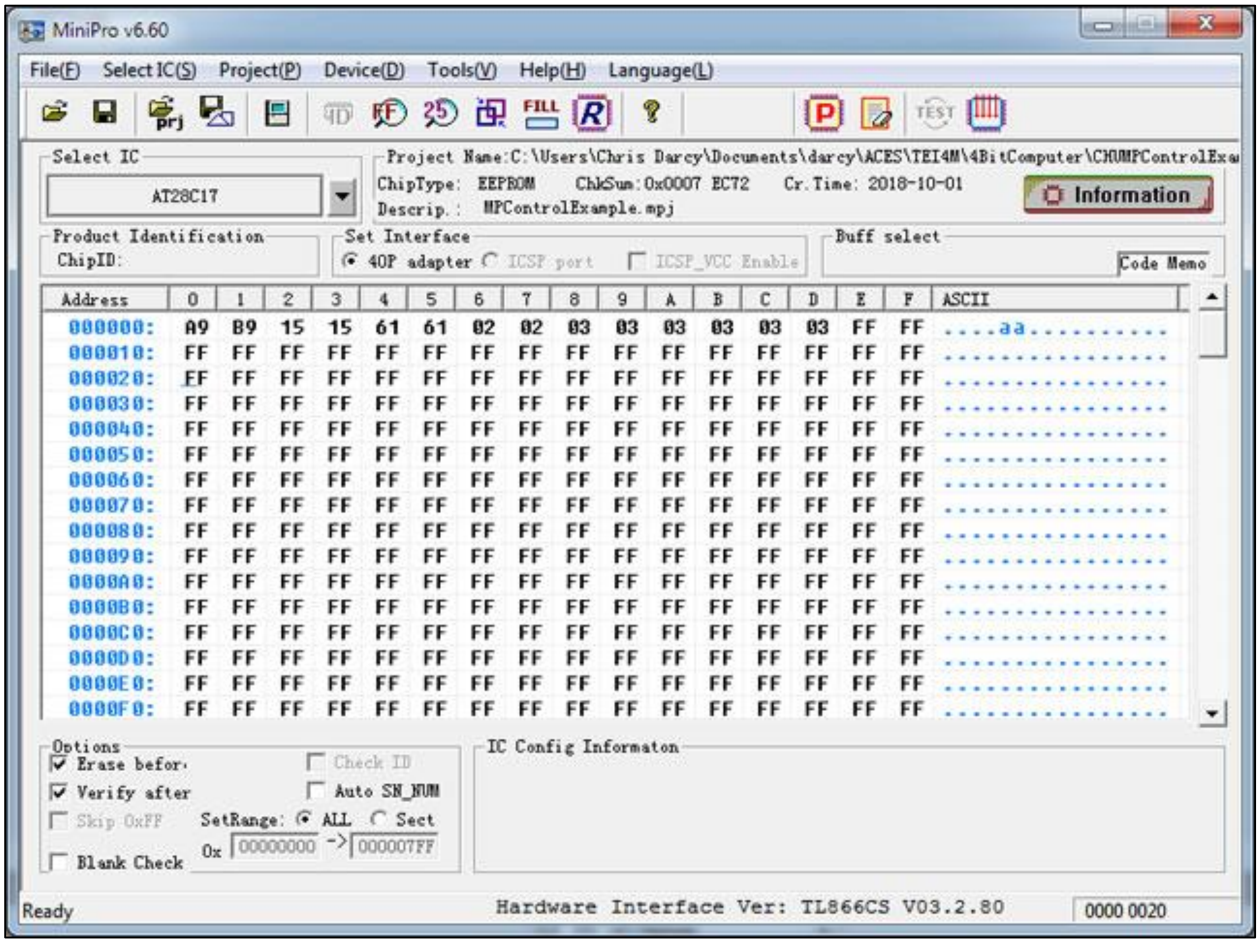
"*With a 4-bit OpCode, we could have as many as 16 distinct operations in our instruction set. Each 4-bit OpCode must be translated into the ten control bits required by our datapath (9 bits from part 1, plus a 10th to control the program counter). Go ahead and write down each instruction's 10 control bit values.*"

Inst	Op 7	Op6	Op5	Op4	Sel	ALU	S3	S2	S1	S0	M	Cn	Acc	RW	PC
LOAD const	0	0	0	0	0	B	1	0	1	0	1	X	0	1	0
LOAD IT	0	0	0	1											
ADD const	0	0	1	0											
ADD IT	0	0	1	1											
SUBTRACT const	0	1	0	0											
SUBTRACT IT	0	1	0	1											
STORETO const	0	1	1	0											
STORETO IT	0	1	1	1											
READ const	1	0	0	0											
READ IT	1	0	0	1											
GOTO const	1	1	0	0											
GOTO IT	1	1	0	1											
IFZERO const	1	1	1	0											
IFZERO IT	1	1	1	1											

"*You may accomplish this translation from 4-bit OpCodes to 10 control bits using combinational logic, or simply by using a Control ROM. Because our ROM outputs 8-bit values, you'll still need to use a little cleverness for those last 2 bits. If you decide to use a ROM, you'll need to determine what 8-bit value to store for each address, and then ask your teacher program these values into a ROM. Either way, you'll need to wire your control logic, and demonstrate to your teacher that you can control the datapath using four loops of wire corresponding to an OpCode.*"

4.3.1 Control Code Sample

Here is a possible set of codes that may need to be adjusted for your build.


MiniPro v6.60
File(F) Select IC(S) Project(P) Device(D) Tools(V) Help(H) Language(L)
Information

```
Select IC: AT28C17
Project Name:C:\Users\Chris Darcy\Documents\darcy\ACES\TEI4M\4BitComputer\CHUMPControlExa
ChipType: EEPROM   ChkSum:0x0007 EC72   Cr.Time: 2018-10-01
Descrip.: MPControlExample.mpj
Product Identification  ChipID:
Set Interface: (•) 40P adapter  ( ) ICSP port  [ ] ICSP_VCC Enable
Buff select: Code Memo

Address    0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F   ASCII
000000:   A9 B9 15 15 61 61 02 02 03 03 03 03 03 03 FF FF   ....aa..........
000010:   FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF   ................
000020:   FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF   ................
000030:   FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF   ................
000040:   FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF   ................
000050:   FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF   ................
000060:   FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF   ................
000070:   FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF   ................
000080:   FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF   ................
000090:   FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF   ................
0000A0:   FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF   ................
0000B0:   FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF   ................
0000C0:   FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF   ................
0000D0:   FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF   ................
0000E0:   FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF   ................
0000F0:   FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF   ................

Options
[✓] Erase befor.      [ ] Check ID
[✓] Verify after      [ ] Auto SN_NUM
[ ] Skip 0xFF         SetRange: (•) ALL  ( ) Sect
[ ] Blank Check       0x 00000000 -> 000007FF
IC Config Informaton

Ready        Hardware Interface Ver: TL866CS V03.2.80        0000 0020
```

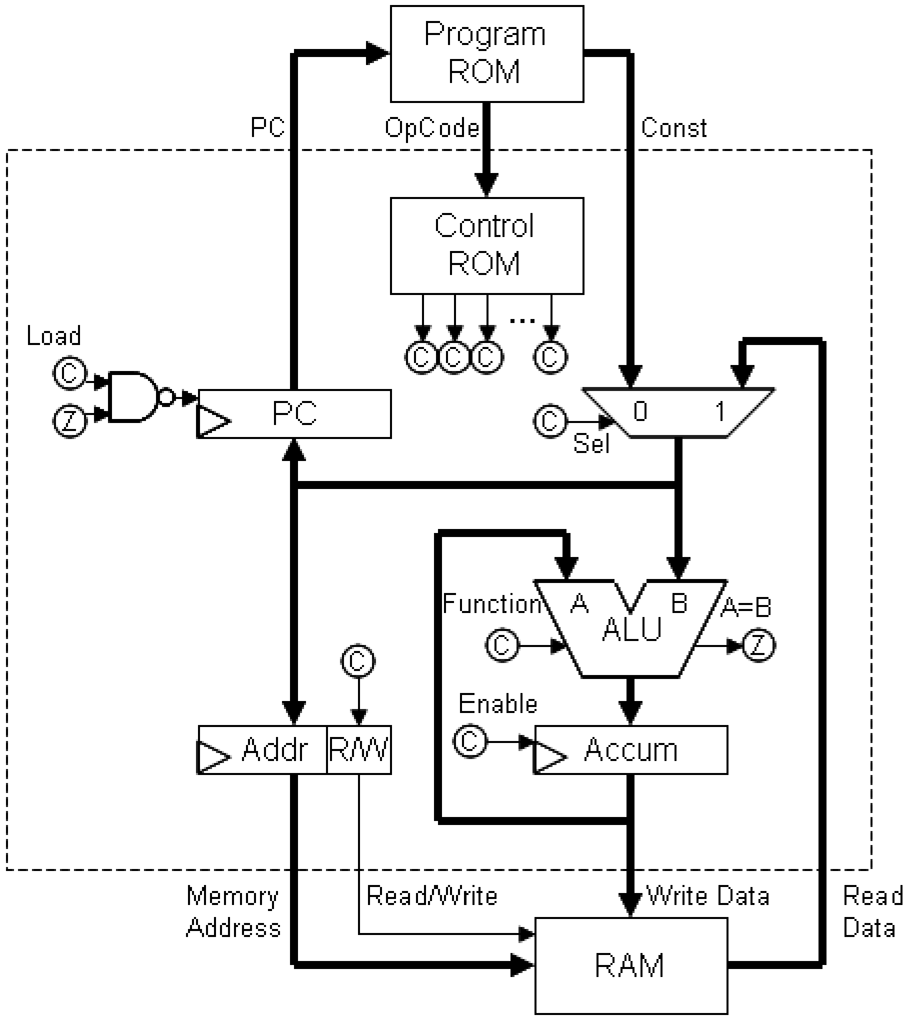
We are now in the home stretch of our 4-bit minimal computer processor build. As simplified as this processor is, it will still consist of over one hundred wires. You will want to double-check each wire, as debugging this project will be rather difficult. No matter how careful you are, you will almost certainly need to go back and debug your work, which is why *neat wiring is essential*. This means that your wires should be measured and bent precisely so that they sit flush against the board. If you do not wire your board neatly, your teacher will not help you debug your work. In other words, *if your wiring is ugly, you're on your own!*

As mentioned previously, you will want to adopt a color convention for your wires and buses. One such convention is to pick a different color for each path. For example, you might use only blue wires to connect the selector output to both the flip-flop inputs and the ALU B inputs. Additionally, you might pick one color for clock signals and another for all control wires.

In the back of this workbook are diagrams of each of the major chips used in this lab. *Before you wire any two pins together, mark the connection on your chip diagrams.* Suppose you are connecting the Program Counter's QA output to the Program EEPROM's A0 input. Next to the Program Counter diagram's QA pin, write "Prog A0" (or similar), and next to the Program ROM's A0 input, write "PC QA". As before, *if you do not mark these diagrams, you're on your own!*

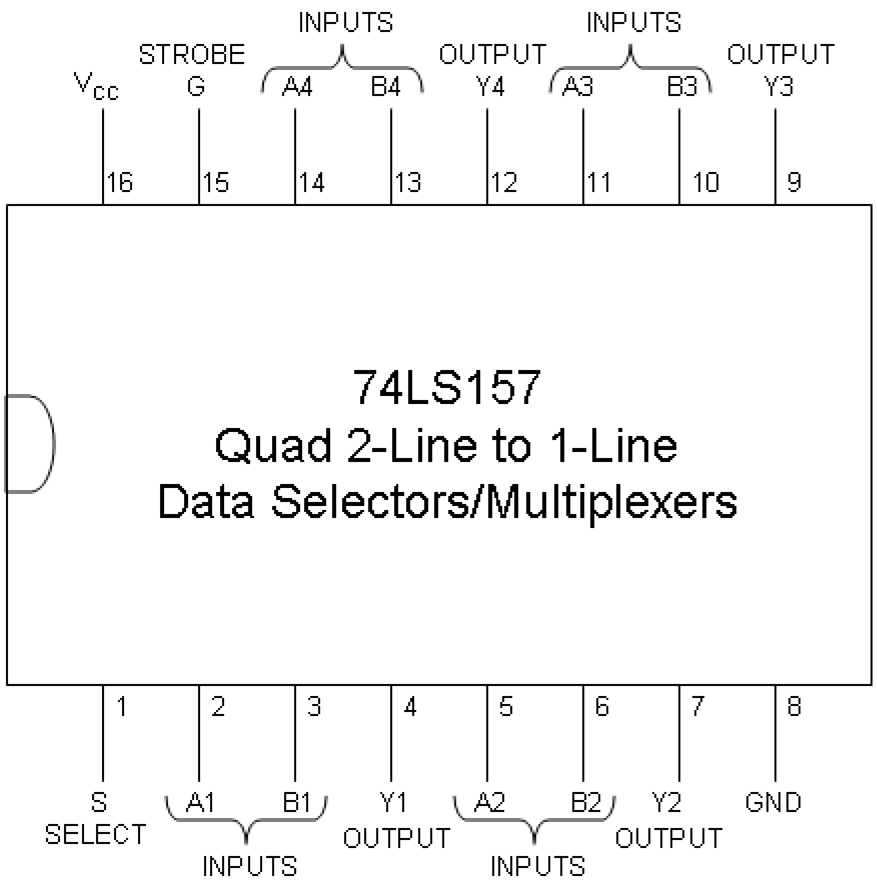
Programming Your CHUMP

Finally, connect the Program ROM in place of the wire loops and switches, as shown below. Then write a simple program that makes use of the various instruction types. Assemble the bits for this program, and ask your teacher to program it into your Program ROM. Finally, demonstrate that your processor runs the program.



5 Reference: CHUMP Chips

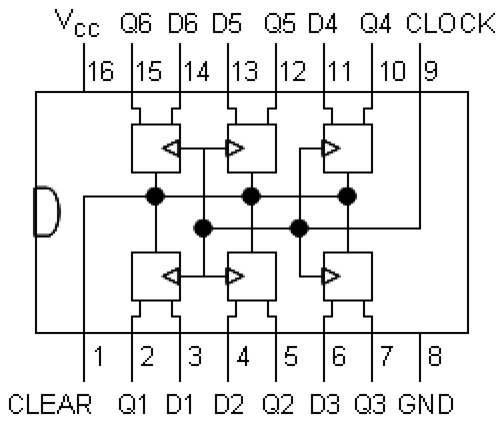
74LS157: Quad 2-Line to 1-Line Data Selectors/Multiplexers



Strobe	Select	A	B	Y
H	X	X	X	**L**
L	L	L	X	**L**
L	L	H	X	**H**
L	H	X	L	**L**
L	H	X	H	**H**

STROBE G should normally be wired low. A high level at the STROBE G input will set all outputs to low.

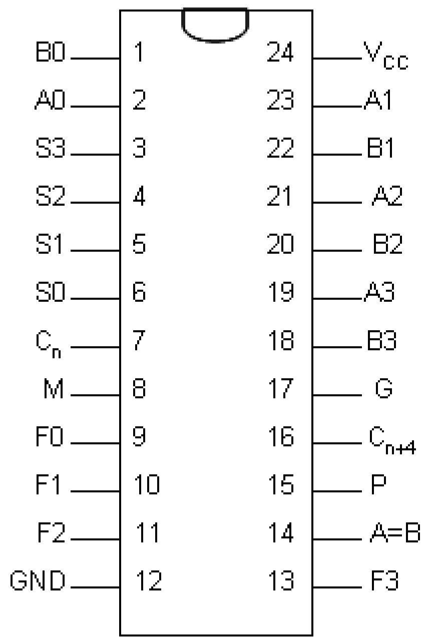
74LS174: Hex D-Type Flip-Flops with Clear



Clear	Clock	D	Q
L	X	X	**L**
H	↑	H	**H**
H	↑	L	**L**
H	L	X	**Q_0**

CLEAR should normally be wired high. A low level at the CLEAR input will immediately set all outputs to low.

74LS181: 4-Bit Arithmetic Logic Unit



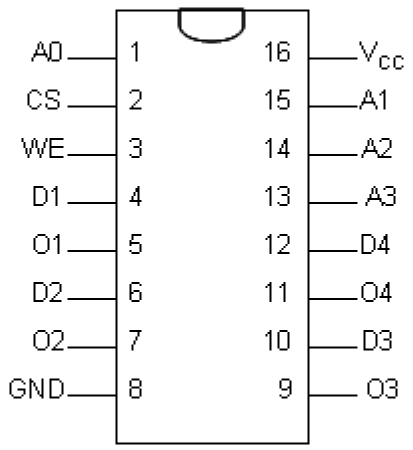
Pin Names	Description
A0 – A3	Operand Inputs
B0 – B3	Operand Inputs
S0 – S3	Function Select Inputs
M	Mode Control Input
C_n	Carry Input
F0 – F3	Function Outputs
A=B	Comparator Output
G	Carry Generate Output
P	Carry Propagate Output
C_{n+4}	Carry Output

The A=B output from the device goes high when all four F outputs are high. The A=B output is open-collector, meaning that it should be connected via a 2.2KΩ resistor to +5 volts.

S3	**S2**	**S1**	**S0**	**Logic:** **(M = H)**	**Arithmetic** **(M = L) (C_n = H)**
L	L	L	L	¬A	A
L	L	L	H	¬A or ¬B	A or B
L	L	H	L	¬A and B	A or ¬B
L	L	H	H	Logic 0	minus 1
L	H	L	L	¬(A and B)	A plus (A and ¬B)
L	H	L	H	¬B	(A or B) plus (A and ¬B)
L	H	H	L	A xor B	A minus B minus 1
L	H	H	H	A and ¬B	(A and B) minus 1
H	L	L	L	¬A or B	A plus (A and B)
H	L	L	H	¬A xor ¬B	A plus B
H	L	H	L	B	(A or ¬B) plus (A and B)
H	L	H	H	A and B	(A and B) minus 1
H	H	L	L	Logic 1	A plus A
H	H	L	H	A or ¬B	(A or B) plus A
H	H	H	L	A or B	(A or ¬B) plus A
H	H	H	H	A	A minus 1

- Arithmetic operations expressed in 2s complement notation.
- In arithmetic mode (M = L), setting C_n = L adds 1 to output.

74S189: 64-Bit Random Access Memory

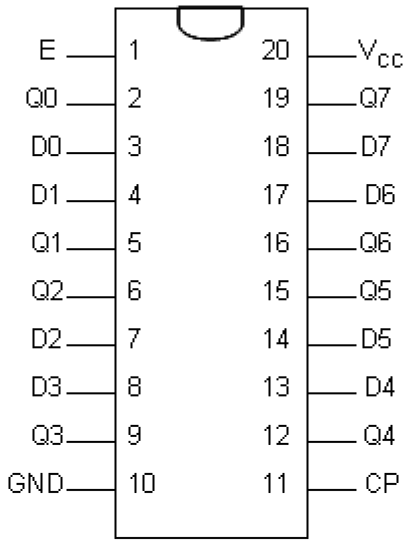


Pin Names	Description
A0 – A3	Address Inputs
CS	Chip Select Input (Active LOW)
WE	Write Enable Input (Active LOW)
D1 – D4	Data Inputs
O1 – O4	Inverted Data Outputs (Open Collector)

Output data is the complement of the stored data. (If you wish to use the output data as is, you'll need to invert data before storing it.)

CS	WE	Operation	Condition of Outputs
L	L	Write	Off
L	H	Read	Complement of Stored Data
H	X	Off	Off

74LS377: 8-Bit Register

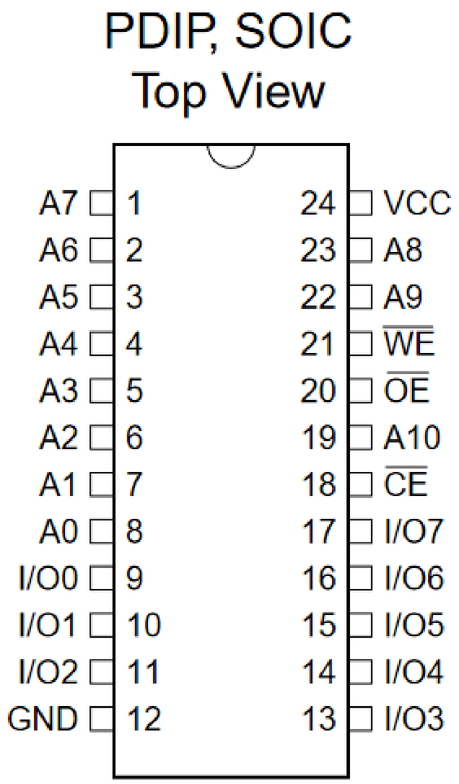


Pin Names	Description
E	Enable Input
D0 – D7	Data Inputs
CP	Clock Pulse Input (Active Rising Edge)
Q0 – Q7	Flip-Flop Outputs

E	CP	Dn	Qn
H	X	X	**No Change**
L	↑	H	**H**
L	↑	L	**L**

AT28C16: 16K (2K x 8) Parallel EEPROM

The CHUMP requires two EEPROM ICs; one for the program and one for the control codes acting as an alternative to complex combinational logic circuitry. Flashing the EEPROMs is where your EEPROM burner comes in. Alternatively, Eater has a terrific video the discussing how to exploit a Nano as an alternative EEPROM burner: https://www.youtube.com/watch?v=BA12Z7gQ4P0



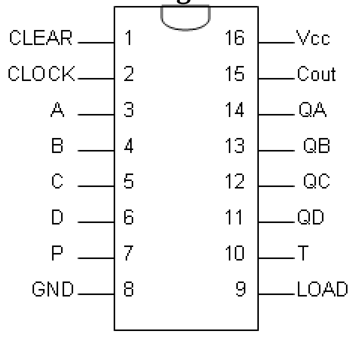
Pin Name	Function
A0 - A10	Addresses
$\overline{CE}$	Chip Enable
$\overline{OE}$	Output Enable
$\overline{WE}$	Write Enable
I/O0 - I/O7	Data Inputs/Outputs
NC	No Connect
DC	Don't Connect

When CE and OE are low and WE is high, the data stored at the memory location determined by the address pins is asserted on the outputs.

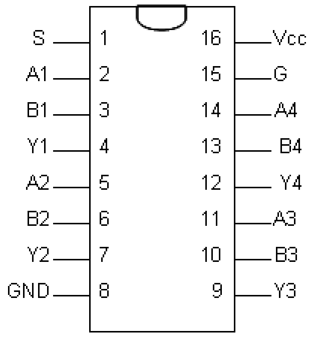
Chip Diagrams

Mark all of your connections on these diagrams.

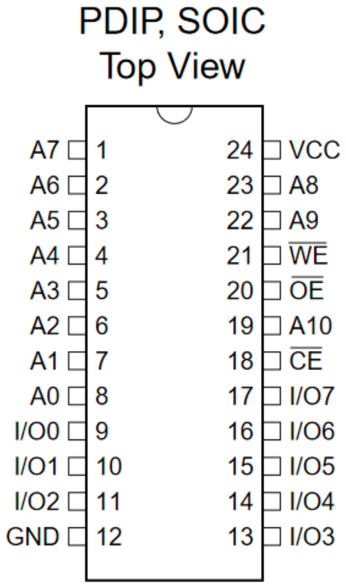
74LS161: Program Counter



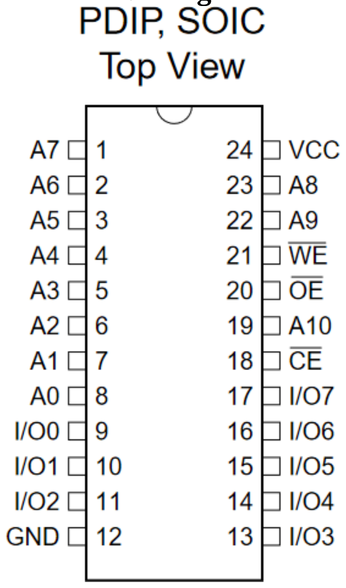
74LS157: Selector



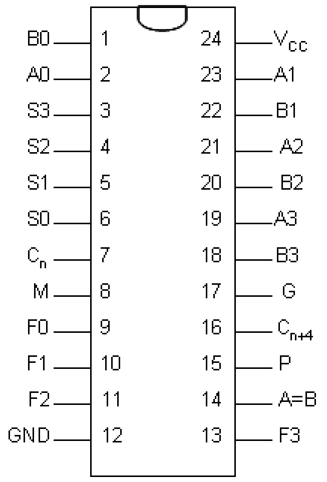
AT28C16: Control ROM



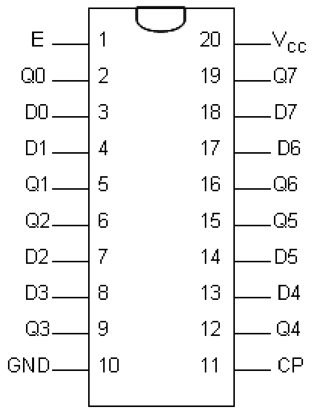
AT28C16: Program ROM



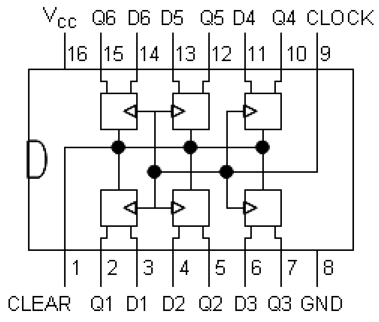
74LS181: ALU



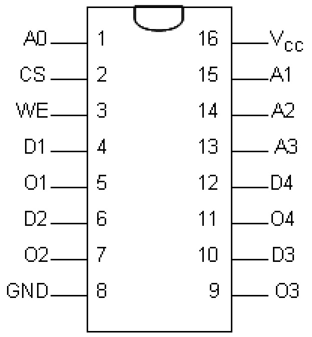
74LS377: Accumulator Register



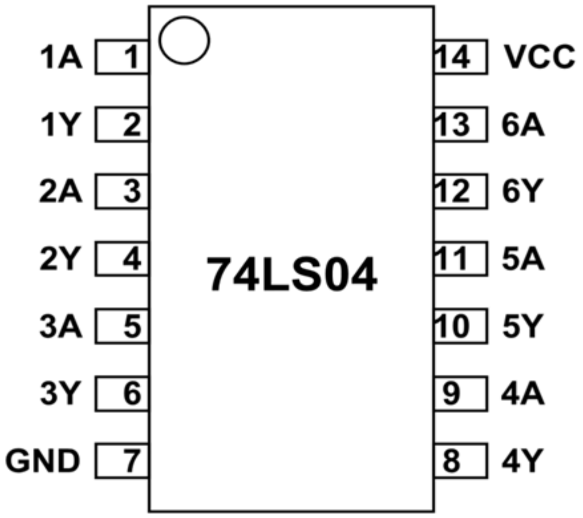
74LS174: Address Flip-Flop



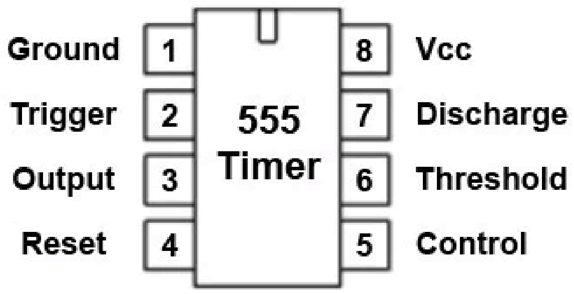
74S189/289: RAM



74LS04: Quad NAND



555: Timer/Counter



Epilog. Notes from D. Feinberg

This final section is a selection of additional handouts from Feinberg for his students. I include them out of respect for his original approach to developing and introducing this project and also to serve as good reminders for ACES.

Lab Rules

By signing this form, the student makes the following promises regarding his/her conduct in the Computer Architecture course.

I will be extremely careful not to create short circuits.

I will use extreme caution when touching potentially overheating components.

I will never engage in dangerous behavior with power adapters, such as sticking wires in my mouth, chaining adapters together, etc.

I will use extreme care in handling sharp objects, such as wire strippers, multimeter probes, and the countless electronic components with small sharp pins.

I will be careful not to force a component into or out of the breadboard.

I will never place another person in danger, whether deliberately or through my negligence. I will never throw anything in the classroom, out the window, etc. I understand that there will be serious consequences for such dangerous behavior.

I will never tamper with another student's lab. I understand that such tampering could set another student's work back many hours, and that any such destruction of property will be dealt with severely.

I understand that, although there is no textbook, there is a lab fee associated with this course. I will inform my parent/guardian that my student account will automatically be charged approximately $65 for lab materials, including wire, a breadboard, a variety of chips, and tools.

Student Signature: ______________________ Date: ________________

Final Feinberg Thoughts

Since writing my "A Simple and Affordable TTL Processor for the Classroom" paper, I had the opportunity to teach the course to another 30 students. In light of that experience, here is some practical advice on how to teach with my lab kit:

1. The most significant change I made the second time I taught the course was in using AC adapters instead of 9V batteries. This eliminated a lot of issues, but it introduced one new one: the voltage regulators would get very hot--even with heat sinks. Since then, my colleague has taught the course a couple of times, and he recommends splurging on regulated 5-volt power supplies, so that you don't have to worry about the over-heating either. [I haven't actually tried getting regulated supplies yet.]

2. Have the students connect an LED that always shows them if their board is on. That way, if they have a short, the light will be off, and they'll know to unplug their board quickly before the regulator overheats.

3. Have the students have a dedicated LED on their board that they can always use as a logic probe. Too many students would build one, test one pin, and then take apart their logic probe every time. Drove me nuts. The logic probe should also include a really long wire, so they can reach any part of their board with it.

4. Near the end of the course, we found that some chips (especially the counters) had very sensitive output pins, so that the mere act of testing if the pin was outputting a high or low voltage would actually change the output. Starting in the NAND lab (I think), my handout used to tell the students to make a logic probe by connecting a long wire to an LED, that to a 330 resistor, and that to ground. But this turns out not to be a great way to test TTL outputs. The better way is to have them connect a long wire to the input of an inverter, the output of the inverter to a 330 resistor, the resistor to an LED, and the LED to 5 volts. The light will normally be on this way, except when the long wire is connected to a pin whose output is 0V. We found this worked much better with those sensitive output pins.

5. Loops of wire connected to 0V or 5V are easier to use than the tiny DIP switches. It looks lame, but most students eventually give up on the DIP switches.

6. Most of the course, you want to use the RS circuit (or something like it) as a manual clock (once you get to finite state machines). I don't think this is explained well in any of my handouts. The circuit is given in one of the handouts, but not how to use it. You want to connect the two inputs to loops of wire that can be easily connected to 0V or 5V. Connect both to 5V normally. Connect the output of the RS circuit to each chip's clock input. Then connect one of the loops of wire to 0V, then back to 5V, then the other to 0V, then back to 5V. That pattern will simulate one clock tick, cycling the output of the RS circuit between 0V and 5V in a nice stable way. I'm sure there's a better way, but I'm not an electrical engineering person.

7. When you do start using the real clock, we found that the clock inputs on the counter chips were also very sensitive. The workaround we found was to connect the output of the clock circuit through an inverter, and then to connect the output of that inverter to the clock inputs on the counters and other chips. That seemed to improve the clock signal.

8. Students were not good at debugging the first time around. I found that it helped to challenge them from the beginning of the course to become expert debuggers and to not ask me for help all the time. It also helped to discuss debugging practices explicitly, and to pose hypothetical debugging problems and ask them how they would go about finding the bug. In the beginning of the course, students tend to just rip out all their wires and rebuild whenever their circuit doesn't work--something you definitely want to discourage. Another thing that students did is that, when they debug, they assume everything's correct, which blinds them to finding whatever they've miswired.

Appendix A. AVR Optimization Toolkit

Below is the list of parts ordered from ABRA.

#	Line	Part	Project	Supplier	Code/Note
1	1	ABRA-48 Breadboard-3220 Tie Points	CHUMP	ABRA	ABRA-48
1	2	5V DC Adapter 2A	CHUMP	ABRA/Adafruit	276-ADA (5V 2A)
1	3	HOOK-UP WIRE SPOOL SET	CHUMP	ABRA/Adafruit	22HW-25-KIT
1	4	Adafruit Wire Strippers/Cutters	CHUMP	ABRA/Adafruit	147-ADA
1	5	12 VDC 1000mA regulated switching power adapter	DC Fan	ABRA/Adafruit	798-ADA
1	6	Arduino Nano Compatible v3 ATMEGA328P	*	ABRA	ARD-NANO
1	7	ALU:74LS181	CHUMP	ABRA	74LS181
2	8	EEPROM:AT28C16	CHUMP	ABRA	28C16A-15
2	9	(Logic) NAND:74LS00	CHUMP	ABRA	74LS00
2	10	(Logic) NOT:74LS04 (Inverter for RAM)	CHUMP	ABRA	74LS04
2	11	(Logic) AND:74LS08 (Branch Control)	CHUMP	ABRA	74LS08
2	12	(Logic) OR: 74LS32 (Clock)	CHUMP	ABRA	74LS32
1	13	MUX/SEL: 74LS157	CHUMP	ABRA	74LS157
1	14	PC: 74LS161	CHUMP	ABRA	74LS161
1	15	ADDRESS: 74LS174	CHUMP	ABRA	74LS174
1	16	ACCUM: 74LS377	CHUMP	ABRA	74LS377
1	17	RAM: 74189	CHUMP	ABRA	74189
4	18	Clock: LM555	CHUMP	ABRA	LM555
6	19	Clock: Momentary Tactile Button (12mm)	CHUMP	ABRA	PBS-190
3	20	Clock: 0.1uF Capacitor	CHUMP	ABRA	753-ADA
4	21	Clock: 1uF Capacitor	CHUMP	ABRA	1R50
4	22	Clock: SPDT Slide Switch	CHUMP	ABRA	SSW-120-BB
20	23	LED 3mm: Red	CHUMP	ABRA	LED-3R
20	24	LED 3mm: Amber	CHUMP	ABRA	LED-3A
20	25	LED 3mm: Green	CHUMP	ABRA	LED-3G