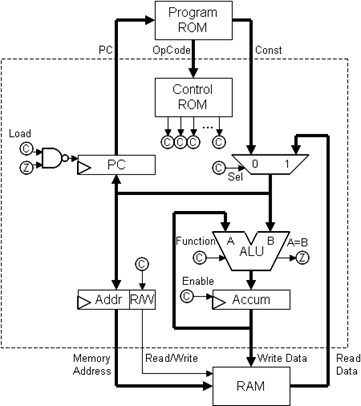
**Project Notes for:**

A Simple and Affordable TTL Processor for the Classroom

David Feinberg  
The Harker School, [fberg@cs.cmu.edu](mailto:fberg@cs.cmu.edu)  
6-2007

****

Course: Computer Science/Engineering

Code: ICS4U

Edited by: C. D’Arcy

Printed: August 27, 2021

**Table of ContenTs**

[0 Introduction 1](#_Toc23394945)

[0.0 Lab: 5 Volts 1](#_Toc23394946)

[0.1 Lab Rules 3](#_Toc23394947)

[0.2 Lab: Transistors 4](#_Toc23394948)

[Resistor Values 4](#_Toc23394949)

[NOT Gate 4](#_Toc23394950)

[NAND Gate 6](#_Toc23394951)

[NOR Gate 6](#_Toc23394952)

[Combinational Logic 7](#_Toc23394953)

[1 Combinational Logic 7](#_Toc23394954)

[1.0 Lab: Combinational Logic 7](#_Toc23394955)

[From Truth Tables to Chips 7](#_Toc23394956)

[Some More Chips 9](#_Toc23394957)

[XOR 10](#_Toc23394958)

[Selectors 11](#_Toc23394959)

[Adders 12](#_Toc23394960)

[Extra Credit Options 13](#_Toc23394961)

[1.1 Lab: NAND Gates 14](#_Toc23394962)

[NAND Gate 15](#_Toc23394963)

[Inverter 15](#_Toc23394964)

[AND Gate 15](#_Toc23394965)

[Another Gate 15](#_Toc23394966)

[NOR Gate 15](#_Toc23394967)

[Additional Credit 15](#_Toc23394968)

[2 Sequential Logic 16](#_Toc23394969)

[2.0 Lab: The Clock 16](#_Toc23394970)

[2.1 Lab: Finite State Machines 17](#_Toc23394971)

[Function Table (Each Flip-Flop): 17](#_Toc23394972)

[A Very Simple Finite State Machine 18](#_Toc23394973)

[2.2 Lab: Sequential Logic 20](#_Toc23394974)

[An RS Circuit 20](#_Toc23394975)

[A Latch 21](#_Toc23394976)

[A Flip-Flop 22](#_Toc23394977)

[3 Memory 23](#_Toc23394978)

[3.0 Lab: Arithmetic 23](#_Toc23394979)

[74LS181: 4-Bit Arithmetic Logic Unit 24](#_Toc23394980)

[Function Table: 25](#_Toc23394981)

[74LS377: Octal D-Type Flip-Flop with Common Enable and Clock 27](#_Toc23394982)

[Pin Descriptions 27](#_Toc23394983)

[Truth Table: 27](#_Toc23394984)

[3.1 Lab: Counters and ROMs 28](#_Toc23394985)

[Making the Counter Intuitive 28](#_Toc23394986)

[When in ROM ... 29](#_Toc23394987)

[4.0 Universality and Programmability 31](#_Toc23394988)

[Game Plan 31](#_Toc23394989)

[5 CHUMP 35](#_Toc23394990)

[5.0 Reference: CHUMP Chips 35](#_Toc23394991)

[74LS157: Quad 2-Line to 1-Line Data Selectors/Multiplexers 35](#_Toc23394992)

[74LS161: Synchronous 4-Bit Binary Counter 36](#_Toc23394993)

[74LS174: Hex D-Type Flip-Flops with Clear 37](#_Toc23394994)

[74LS181: 4-Bit Arithmetic Logic Unit 38](#_Toc23394995)

[74S189: 64-Bit Random Access Memory 40](#_Toc23394996)

[74LS377: 8-Bit Register 41](#_Toc23394997)

[28C17: 16K (2K x 8) Parallel EEPROM 42](#_Toc23394998)

[5.1 Lab: CHUMP 43](#_Toc23394999)

[The CHUMP Instruction Set 44](#_Toc23395000)

[Part 1: The CHUMP Datapath 46](#_Toc23395001)

[Part 2: CHUMP OpCodes 47](#_Toc23395002)

[Chip Diagrams 50](#_Toc23395003)

[6. Chumpanese 52](#_Toc23395004)

[6.0 Chumpanese Guidelines 52](#_Toc23395005)

[6.1 Written Lab: Chumpanese 56](#_Toc23395006)

[6.2 Worksheet: Chumpanese 64](#_Toc23395007)

[Review 64](#_Toc23395008)

[Objects 66](#_Toc23395009)

[Linked Lists 70](#_Toc23395010)

[Arrays of Primitives 71](#_Toc23395011)

[Arrays of Objects 72](#_Toc23395012)

[Stacks 72](#_Toc23395013)

[A First Pass at Procedures 73](#_Toc23395014)

[Returning from Procedures 74](#_Toc23395015)

[The Procedure Stack 75](#_Toc23395016)

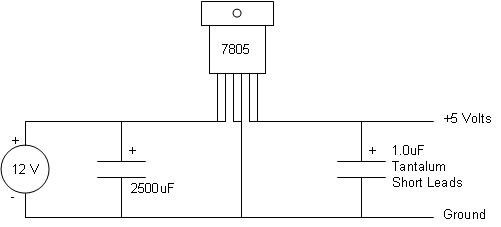
[Recursion 79](#_Toc23395017)

[Email Notes from D. Feinberg 82](#_Toc23395018)

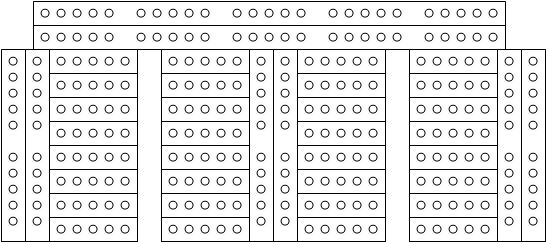
# 0 Introduction

## 0.0 Lab: 5 Volts

The purpose of this lab is simply to use a not-so-steady 12-volt AC/DC power adapter to create a steady 5-volt power supply that we can use to power all our digital components throughout the course. To accomplish this, we'll use a 5-volt regulator (7805) and a couple of capacitors, connected as shown below.



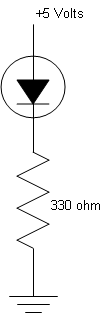
It will take some ingenuity to figure out how to lay this circuit out on your breadboard. Beneath the surface of your breadboard, the little holes are connected by strips of metal in the following arrangement. When you plug in one lead from each of two components above the same metal strip, the components will be connected in your circuit.



Make sure you pay attention to which leads are positive/negative for each component—including the capacitors. You'll probably want to keep your components very close together in the upper right corner of your board, since we'll be adding a lot more components to the board throughout the course.

***Do not plug in your power adapter.*** Instead, call your teacher over to check your wiring. Your teacher will verify with you that your output voltage is a steady 5 volts.

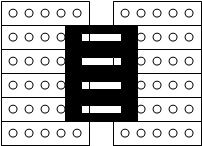
From now on, we will only be concerned with the 5-volt side of the circuit, and we'll ***never*** touch the 12-volt side. If your lab kit ever sparks, or any component becomes dangerously hot, unplug it immediately and let your teacher know! Be especially careful about touching the metal top of the 7805 regulator, which may become hot enough to burn you if you accidentally short-circuit your kit.

Connect 2 of the long rows on the top of your board and all of the long columns so that red marks indicate +5 volts and blue/black marks indicate 0 volts (ground).

When you have done that, build the following circuit. You must connect the LED so that the long positive lead on the round side of the LED is facing +5 volts, and the short negative lead on the flat side of the LED is facing ground.

When connected to your 5-volt power supply, the LED (light-emitting diode) will light up. Throughout the course, we'll use this circuit as a logic probe to test whether a particular wire is carrying a 1 or a 0. A 1 (5 volts) will cause the LED to light up, while a 0 (0 volts) will not.

Next, place a DIP switch in your board as shown below. When a switch is "on", the metal strip below the 5 holes on one side of the switch will be connected to the metal strip below the 5 holes on the other side. When the switch is "off", the two metal strips are no longer connected. Connect one of the switches so that the LED lights up only when the switch is on.



When you've got this working, try modifying your circuit so that the LED lights up only when a certain switch is on, and either one of two other switches. Then ask your teacher to check off your lab.

## 0.1 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: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

## 0.2 Lab: Transistors

### Resistor Values

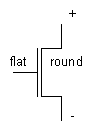
In this lab, you'll be using a number of resistors with different resistance values. You can determine the value of a resistor by looking at the colored bands. Hold the resistor so that the silver or gold band is on the right side, and read off the three remaining colors from left to right. Each color corresponds to a number. The first few of these are listed below.



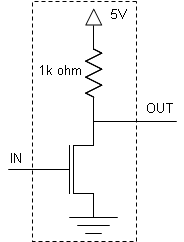
black = 0 brown = 1 red = 2 orange = 3

The first two colors will give you the first two digits of the value, and the third will tell you how many zeroes should follow. For example, if the resistor had colors orange (3), black (0), red (2), then its value can be written out as 3 0 followed by 2 zeroes—in other words, 3000, or 3k ohms.

### NOT Gate

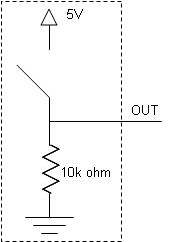
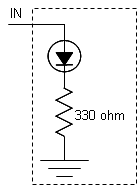
For this lab you will need four transistors. We'll use the following schematic diagram to represent a transistor.

We can use a transistor to build the following NOT gate.

*schematic for NOT gate symbol for NOT gate*

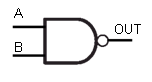
Go ahead and build a NOT gate on your breadboard. Wire the output of the gate to the output indicator circuit shown below (which you'll recall from the previous lab). Try connecting the input of the NOT gate to ground, and then try connecting the input to 5 volts instead. Does the LED light up appropriately? Now try connecting the input of the NOT gate to the output of the input switch circuit shown below.

*input switch circuit output indicator circuit*

### NAND Gate

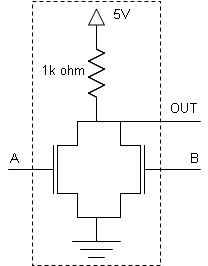
Add a second transistor to your circuit to build a NAND gate circuit as shown below. Connect your first two switches to the inputs of the NAND gate, and connect the output to an LED.

*schematic for NAND gate symbol for NAND gate*

### NOR Gate

When you have tested that your NAND gate works, go ahead and build the NOR gate shown below *without destroying your NAND circuit*. Connect your second two switches to the inputs of your NOR gate, and connect the output to a second LED.

*schematic for NOR gate symbol for NOR gate*

## Combinational Logic

When you are satisfied that both your NAND and NOR gates work correctly, modify your circuit to correspond to the following diagram, with three input switches and one output LED. When you have tested that your circuit behaves appropriately, show your teacher to get your lab checked off.

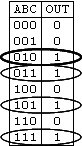
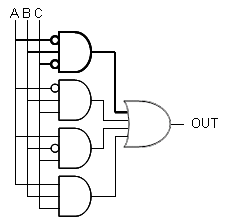


# 1 Combinational Logic

## 1.0 Lab: Combinational Logic

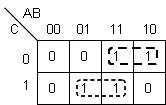
### From Truth Tables to Chips

Suppose we want to build a combinational logic circuit corresponding to the following truth table (whose output is 1 only when the binary value ABC corresponds to a prime number). One systematic way to do this is to identify each row where the output is a 1, and represent it using some inverters (shown as bubbles) and a multi-input AND gate (which can be constructed from 2-input AND gates). For example, the 010 row is represented by the top AND gate, which will only output a 1 when A=0, B=1, and C=0. We connect the outputs of the AND gates to a single OR gate, so that our output will be 1 when the input is 010, 011, 101, *or* 111.

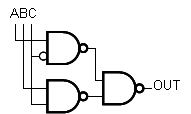
*Truth Table "Brute Force" Sum-of-Products Circuit*

Although this "brute force" method is fairly straight-forward, the resulting circuit is unnecessarily complex. We can find a simpler and equivalent circuit by rewriting the table as a Karnaugh Map, as shown below. In a Karnaugh Map, moving horizontally or vertically to an adjacent box (including moving between the 00 and 10 columns) represents changing exactly one input—a property which will help us see how to construct circuits with fewer gates. Having filled in the map, we identify regions of 1s using as few 2x2, 1x2, 2x1, and 1x1 boxes as possible. As before, each of these regions will correspond to an AND gate in our circuit. In the map below, the upper region corresponds to the situation when A=1 and C=0, and is thus transformed into the top AND gate in our circuit below.

*Karnaugh Map Minimal Sum-of-Products Circuit*

One more neat trick. We can invert the outputs of the AND gates and the inputs of the OR gate without changing the behavior of the circuit. It is now easy to see that the resulting circuit is equivalent to one in which we replace all the AND and OR gates by NAND gates.

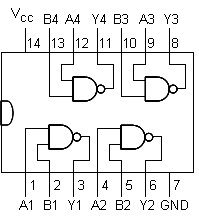
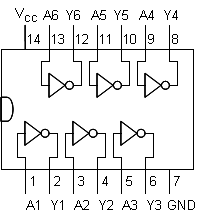
 

*With Extra Inverters NAND-NAND Arrangement*

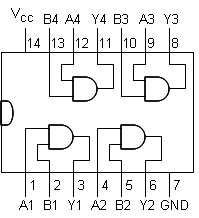
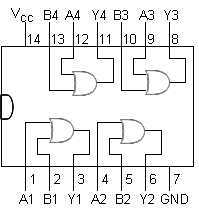
Although circuits are often built using this NAND-NAND arrangements, we can use the more intuitive AND-OR arrangement in our lab, thanks to ...

### Some More Chips

Now that you've succeeded in building AND, OR, and NOT gates using the 74LS00 NAND chip, you have earned a few more chips for your collection.

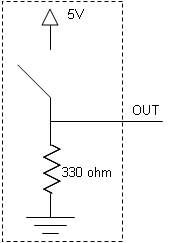
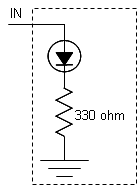
*74LS00 NAND Chip 74LS04 Inverter Chip*

*74LS08 AND Chip 74LS32 OR Chip*

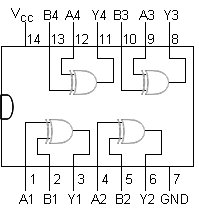
### XOR

In this lab, you will use these chips to construct higher level combinational logic circuits. Your first task is to build an XOR ("exclusive or") circuit—one whose output is 1 only when exactly one of its two inputs is a 1. Here, you will find the Karnaugh Map strategy does no better than the "brute force" sum-of-products strategy. Connect the output to an LED and each input to a switch, using the standard switch and LED circuits shown below.

*Switch Circuit LED Circuit*

When you have completed your circuit, demonstrate it to your teacher. By doing so, you will earn an XOR chip, which you may find helpful in building other circuits in this lab.

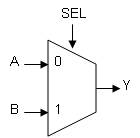
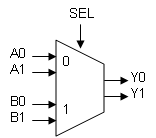
 

*XOR Gate Symbol 74LS86 XOR Chip*

### Selectors

Design a *hazard-free* 1-bit selector circuit (also called a "multiplexer", or "mux"). When the SEL input is 0, the output Y will match the value of the A input. When the SEL input is 1, the output Y will match the value of the B input. What programming construct does the selector resemble?

Use your design to build a hazard-free 2-bit selector, in which Y0 and Y1 correspond to A0 and A1 when SEL is 0, and correspond to B0 and B1 when SEL is 1. Connect your 2-bit selector to five switches (in some meaningful arrangement) and two LEDs. Show your design to your teacher, and demonstrate that it works correctly.

*A 1-Bit Selector A 2-Bit Selector*

### Adders

Design a circuit that adds two 1-bit inputs together, and uses a 2-bit output for the sum. In other words, if the inputs are 0 and 0, the output should be 00 (zero). If the inputs are 0 and 1 (in either order), the output should be 01 (one). If the inputs are 1 and 1, the output should be 10 (two). (Hint: Consider each output bit separately.)

Now design and build a circuit that adds *three* 1-bit inputs together, and uses a 2-bit output for the sum. Connect 3 switches and 2 LEDs, and demonstrate your working circuit to your teacher.

It turns out that multi-digit binary numbers can be added using exactly the same algorithm you learned for adding base-10 numbers. Namely, add each column beginning with the least significant digit, being sure to carry a 1 to the next column whenever the sum is 10 (two) or higher.

1

1 1 0

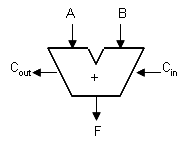
+ 1 1 1

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1 1 0 1

*Computing 6 + 7 = 13*

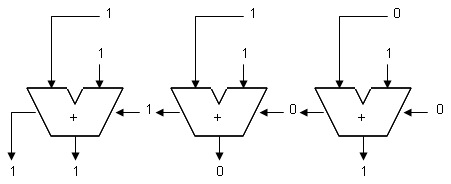
Suppose we wish to build a circuit to add a single column. It will require three inputs—a digit from the first addend A, a digit from the second addend B, and the bit carried from the last column Cin. Furthermore, each column has two outputs—the sum F, and the bit to carry to the next column Cout.



*Circuit for Adding One Column*

But here's the punchline: This is the circuit you just built! It takes three 1-bit inputs, adds them together, and outputs the 2-bit sum. You only need to choose one input to be Cin, to call the less significant output bit F, and the more significant output bit Cout.

We can now connect *n* adder circuits to add 2 *n*-bit numbers, as shown here.



*Using 3 adders to compute 6 + 7 = 13*

Find another student with a working adder, and connect them together to add 2-bit (or higher!) binary numbers. Show this combined circuit to your teacher.

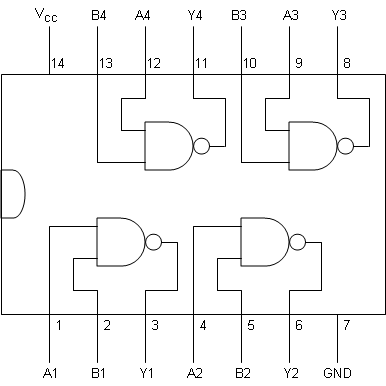
### Extra Credit Options

If you finish early, talk to your teacher about some of the following project ideas.

* Display a 3-bit binary number using LEDs arranged like the pips on a die.
* Display a 3-bit binary number using a 7-segment display.
* Create a circuit for subtracting multi-bit numbers.
* Create a circuit for multiplying multi-bit numbers.
* Create a 2-bit arithmetic logic unit.
* Invent your own combinational logic device!

## 1.1 Lab: NAND Gates

In this lab, you will use the 74LS00 NAND chip, which contains 4 2-input NAND gates on an integrated circuit (IC). A connection diagram for the chip is shown below.



*74LS00 Connection Diagram*

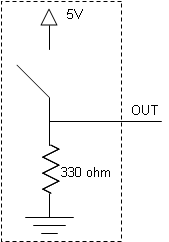
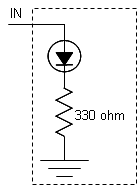
Place the chip in your breadboard so that the notch points toward the top of your board—a convention we'll adopt for the rest of the course. Make sure that all the pins line up with the holes on your board, so that none of them get bent when you insert the chip.

The pin immediately to the left of the notch is pin 1, and the numbers increase as you go around the chip counterclockwise. This same numbering scheme is used by all chips.

This chip belongs to a family of ICs we'll be using throughout the course called TTL chips. On all of these chips, when you view the chip so that the notch is at the top, the ground pin will be on the lower left and the VCC pin will be on the upper left. The ground pin connects (indirectly) to the emitter (-) terminal of each of the transistors on the IC. The VCC pin connects (indirectly) to the collector (+) terminal of each of the transistors. (The "CC" stands for "common collector".) You should therefore connect the ground pin to ground (or 0 volts) on your breadboard, and connect the VCC to +5 volts.

### NAND Gate

Connect Y1 to the LED circuit we used in earlier labs. Try connecting A1 and B1 to high (+5) or low (ground) to test the NAND gate. Does the LED light up when you expect it to? Test what happens when an input is not connected to anything. Do disconnected TTL input pins behave as if they are connected to high or to low? Now connect two switches to A1 and B1, so that you can control the LED with the switch. Here are the switch and LED circuits to use. (Note that the resistor value on the switch circuit is different from the previous lab We'll use 330Ω resistors whenever we use switches as inputs to TTL chips.)

*Switch Circuit LED Circuit*

### Inverter

Disconnect the switch from B1, and connect B1 in such a way that the LED is on only when A1's switch is off. If you succeed, you'll have created a NOT gate (often called an "inverter") from a NAND gate!

### AND Gate

Using what you have just learned, how can you use two NAND gates to create a single AND gate? Implement this 2-NAND-gate idea, so that the LED lights up only when both switches are on.

### Another Gate

What happens if you invert both inputs to a NAND gate? Using three NAND gates, go ahead and build this gate. Does the LED behave the way you predicted?

### NOR Gate

Finally, use all 4 NAND gates on the chip to create a single NOR gate. To get your lab checked off, demonstrate to your teacher that the LED lights up appropriately.

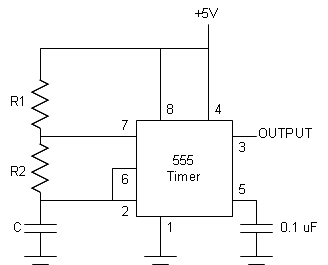
### Additional Credit

Once you have been checked off for this lab, earn additional credit by building a 3-input AND gate, so that the LED is on only when all three switches are on.

# 2 Sequential Logic

## 2.0 Lab: The Clock

In this quick lab, you will *permanently* wire a clock circuit onto your breadboard. Because this circuit will remain on your board for the rest of the course, be sure your wiring is neat and *compact*.



*The Clock Circuit*

We'll use a 1K ohm resistor for R1 and a 0.47 µF capacitor for C. The resistor for R2 can range from 1K to 3.3M ohms, depending on how fast we want our clock to tick. The *period* of the square wave that our clock will output is given by the following formula.

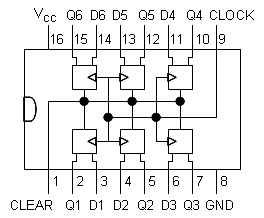
0.693 (R1 + 2R2) C

In this lab, you'll use a 1.5M ohm resistor for R2. What will be the period of your clock? How many times per second (Hertz) will your clock tick?

Connect the output of your clock to an LED circuit, to test that it behaves as expected. (You'd need an oscilloscope to know for sure that it was giving you a square wave.)

## 2.1 Lab: Finite State Machines

For this, lab you'll need a 74LS174 chip, shown in the following connection diagram.



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

### Function Table (Each Flip-Flop):

|  |  |  |  |
| --- | --- | --- | --- |
| Clear | Clock | D | **Q** |
| L | X | X | **L** |
| H | ↑ | H | **H** |
| H | ↑ | L | **L** |
| H | L | X | **Q0** |

H = HIGH LEVEL (steady state)

L = LOW LEVEL (steady state)

X = Don't Care

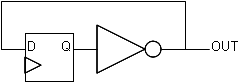
↑ = Transition from LOW-to-HIGH level

Q0 = The level of Q before the indicated steady-state input conditions were established

In other words the Clear pin must be wired HIGH to enable the flip-flops.

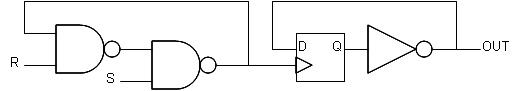
### A Very Simple Finite State Machine

Go ahead and build the following very simple Finite State Machine.

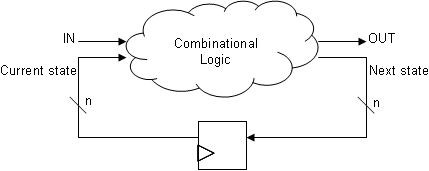


Connect the output to an LED, and the clock input to a switch. What happens when you toggle the switch?

Try connecting the clock input to the output of the RS circuit, as follows, so that you can manually clock the circuit with one pulse at a time.



This Finite State Machine has only two states, takes in no input, and its output is exactly the same as its state number. In general, our FSMs will look like the following. If we have *n* state bits, how many different states can our FSM have?



Typically, we'll use the state number 00...0 as the initial state of our machine. If you connect one of your switches to the flip-flop chip's clear pin, you can reset your machine back to its initial state by turning the switch off, and start it by turning the switch on.

Try building a few of the following state machines. Whenever you get one working, demonstrate it to your teacher. Some of these will work better when manually clocked using the RS circuit, and others will prefer to have input from a real clock signal.

**Difficulty 1**

Make an FSM that lights up an LED only if it's seen an even number of 1s.

**Difficulty 1**

Make a 2-bit binary counter, that lights up two LEDs in the sequence 00, 01, 10, 11, 00, etc.

**Difficulty 2**

Make an FSM that lights up if it has *ever* seen the input sequence 010.

**Difficulty 2**

Make an FSM that lights up an LED only if the stream of 0s and 1s it has read in encode for a multiple of 3 in binary. For exame, if your machine has seen 1, then 1, then 0, then it has seen 110, which is 6 in binary. Because 6 is a multiple of 3 in binary, your LED will be on. If it then sees a 1, it will have seen 1101, which is 13 in binary, so your LED will be off.

**Difficulty 3**

Make a circuit that could be used like a buzzer in a game show. It should have 2 inputs, and should light up 1 LED if switch 1 turns on first, and a different LED if switch 2 turns on first.

**Difficulty 4**

Make an FSM that works as follows. If the first input it sees is 0, then it will only light up in the future when it has just seen a 0. Otherwise, it will light up only when it has seen a 1.

**Difficulty 5**

Make an FSM that lights up only if it has seen an even number of 1s *and* it has at some point seen two consecutive 0s.

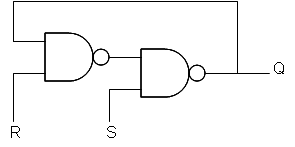
**Difficulty 5**

Make a 2-bit binary counter with a load input. If the load input is HIGH, the counter will proceed to the next state. If the load input is LOW, the counter will go next to whatever state is indicated by two other inputs. In other words, if the two inputs are 10 and the load input is LOW, then the counter will go to state 10 and continue counting, regardless of what state it was in before.

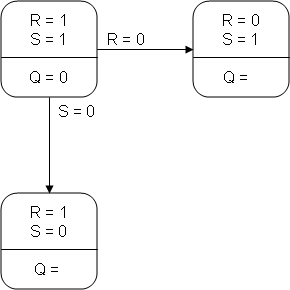
## 2.2 Lab: Sequential Logic

### An RS Circuit

Connect two NAND gates as follows. This circuit will be so useful to us that you should wire it in a neat, permanent fashion in some corner of your board.



Connect the R and S inputs to switch circuits, and Q to an LED circuit. How does Q respond to changes in R and S? Carefully **complete** the following state transition diagram to show how this circuit behaves, adding as many states and transitions as necessary.



Now, suppose we disallow any state where R and S are both 0, along with any transitions to such a state. In the remaining state diagram, how does R earn the name "Reset" and S earn the name "Set"?

Using this idea, make sure you can "set" and "reset" the output LED.

### A Latch

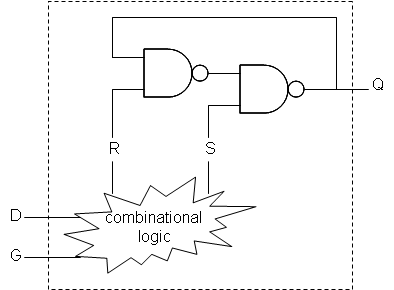
We would like to use this idea to build a latch, represented by the following symbol.



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 | QOLD | **Q** |
| 0 | X | 0 | **0** |
| 0 | X | 1 | **1** |
| 1 | 0 | X | **0** |
| 1 | 1 | X | **1** |

We can build this from the RS circuit by inserting some combinational logic between the inputs G and D, and the R and S inputs.



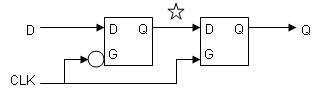
Fill in the following truth table to help you determine what logic gates are needed between D/G and R/S to help us build our latch.

|  |  |  |  |
| --- | --- | --- | --- |
| G | D | **R** | **S** |
| 0 | 0 |  |  |
| 0 | 1 |  |  |
| 1 | 0 |  |  |
| 1 | 1 |  |  |

Go ahead and build the latch in this manner, connecting your switches to the G and D inputs (instead of R and S), and its output Q to the LED. Does it behave correctly?

### A Flip-Flop

*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 tollboth 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.

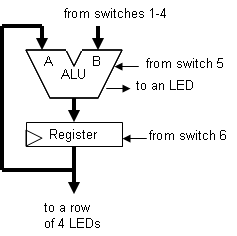
# 3 Memory

## 3.0 Lab: Arithmetic

In this lab you will use two new chips. One of these is the 74LS181 4-bit Arithmetic Logic Unit (ALU), a complex combinational logic circuit capable of performing various arighmetic operations. The other is the 74LS377 8-bit register, which you can think of as 8 flip-flops that can be configured to ignore their input.

This will be the first real test of your neat wiring technique. Somewhere on your board (probably near the center), use a single color of wire to connect permanently (and therefore *neatly*) the ALU's function outputs to four of the register's data inputs.

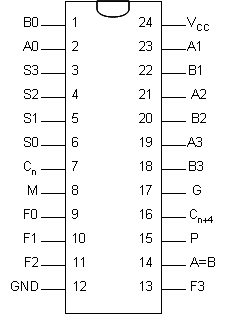
Then, use another color wire to connect the register's corresponding outputs back to the A operand inputs of the ALU. Using the same color, permanently connect the output of the register to an ordered row of 4 LEDs.



Next, temporarily wire switches 1-4 to input B of the ALU, and switch 6 to the enable input on the register. We'll use switch 5 to control which arithmetic operation the ALU performs. When switch 5 is low, the ALU should simply output "B"—the value from switches 1-4. When switch 5 is high, the ALU should output "A minus B". Finally, connect the ALU's A=B output to an LED.

### 74LS181: 4-Bit Arithmetic Logic Unit

Connection Diagram:



Pin Descriptions:

|  |  |
| --- | --- |
| Pin Names | Description |
| A0 – A3 | Operand Inputs |
| B0 – B3 | Operand Inputs |
| S0 – S3 | Function Select Inputs |
| M | Mode Control Input |
| Cn | Carry Input |
| F0 – F3 | Function Outputs |
| A=B | Comparator Output |
| G | Carry Generate Output |
| P | Carry Propagate Output |
| Cn+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.

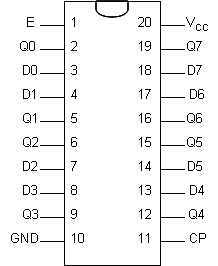
### Function Table:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| S3 | S2 | S1 | S0 | Logic  (M = H) | Arithmetic  (M = L) (Cn = 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 Cn = L adds 1 to output.

### 74LS377: Octal D-Type Flip-Flop with Common Enable and Clock

Connection Diagram:



### Pin Descriptions

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

### Truth Table:

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

H = HIGH Voltage Level

L = LOW Voltage Level

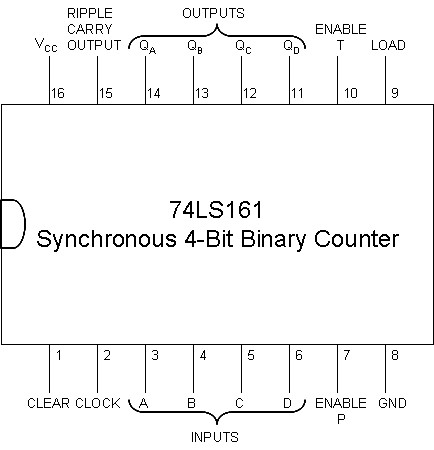
X = Immaterial

## 3.1 Lab: Counters and ROMs

### Making the Counter Intuitive

You now know how to implement a counter as a finite state machine. This device is so useful that we can buy a counter chip, shown below. Here are some things you should know about the 74LS161 counter chip you'll be using.

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

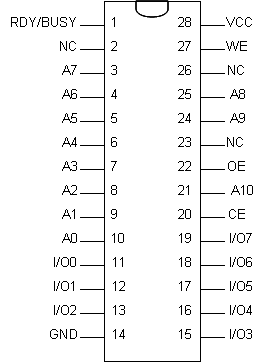


Note that the counter and ROM chips we'll be using in this lab will have a *permanent* home on our boards. It is recommended that you place the counter in the upper left corner of your board, and eventually place the ROM immediately to the right of the counter. Wires to switches and LEDs will be temporary,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

Go ahead and connect the inputs to four switches, and the outputs to a row of 4 LEDs. Now verify that your LEDs begin counting correctly, and that you can load the input value as the count. Demonstrate this to your teacher, and you'll earn your ROM chip.

### When in ROM ...

A ROM is simply a hard-wired truth table. We can think of such a truth table as a kind of memory, since it remembers the output values associated with any given input values. We can therefore think of the input values as *addresses*, which identify output *data* values. The following diagram shows the pins for the ROM chip we'll be using: the 28C17 EEPROM (Electrically Erasable and Programmable Read Only Memory).



Pin Configurations:

|  |  |
| --- | --- |
| Pin Name | Function |
| A0 – A10 | Addresses |
| CE | Chip Enable |
| OE | Output Enable |
| WE | Write Enable |
| I/O0 – I/O7 | Data Inputs/Outputs |
| RDY/BUSY | Ready/Busy Output |
| NC | No 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.

The ROM chip you've been given has some data already stored in it. Your challenge is to determine those values. The data has been stored at addresses 00000000000 through 00000001111, where A0 is the low-order bit. The output values range from 00000000 to 00001111, where I/O0 is the low-order bit. Leaving your switches connected to the counter, connect the counter's output pins to the ROM's address pins. Then connect the ROM's output pins to your row of LEDs. Now, as the counter chip counts up, the addresses will increase, and the LEDs will light up with the values stored in the ROM. To get checked off for this lab, you must correctly identify the 16 numeric values in the ROM.

# 4.0 Universality and Programmability

## Game Plan

1. Turing – must cover now

2. FSM as controller – must cover now

3. RAM

4. The CHUMP: Cheap Homebrew Understandable Minimal Processor

Big questions:

* Given any computational task, can I design an FSM to solve it?
* Is there a single machine that can perform every task?

Introducing Alan Turing

* when was the computer invented? what did the word "computer" mean before that?
* who are the important figures in computer science?
* anyone ever hear of alan turing? what did he do?
* **1912**–**1954** in **Britain.**
* widely considered to be the **father of computer science**
* formalized the concept of "algorithm" and "computation" using an idea called a Turing machine, described in his famous 1936 paper: "On Computable Numbers, with an Application to the Entscheidungsproblem".
* one of the most important outstanding problems in mathematics in the early 1900s was known as the "Entscheidungsproblem" ("decision problem", in german)
* decision problem: find a general algorithm which examines a given mathematical statement and decides if it is universally true or not.
* for example, if there were such an algorithm, i could give it the statement "every even number starting with 6 is the sum of 2 prime numbers", and this algorithm would tell me if this statement were true or false.
* in his early 20s, working alone, Turing proved that such an algorithm does not exist. how did he do that?
* turing spent his whole life thinking about how the mind might work. in thinking about how the mind executes an algorithm to solve a math problem, Turing came up with the idea of a Turing Machine.

FROM TURING'S PAPER (or read from hodges book)

We may compare a man in the process of computing a real number to a machine which is only capable of a finite number of conditions q1, q2, ..., qR which will be called “m-configurations”. The machine is supplied with a “tape”, (the analogue of paper) running through it, and divided into sections (called “squares”) each capable of bearing a “symbol”. At any moment there is just one square, say the r-th, bearing the symbol S(r) which is “in the machine”. We may call this square the “scanned square”. The symbol on the scanned square may be called the “scanned symbol”. The “scanned symbol” is the only one of which the machine is, so to speak, “directly aware”. However, by altering its m-configuration the machine can effectively remember some of the symbols which it has “seen” (scanned) previously. The possible behaviour of the machine at any moment is determined by the m-configuration qn and the scanned symbol S(r). This pair qn, S(r) will be called the “configuration”: thus the configuration determines the possible behaviour of the machine. In some of the configurations in which the scanned square is blank (i.e. bears no symbol) the machine writes down a new symbol on the scanned square: in other configurations it erases the scanned symbol. The machine may also change the square which is being scanned, but only by shifting it one place to right or left. In addition to any of these operations the m-configuration may be changed. Some of the symbols written down {232} will form the sequence of figures which is the decimal of the real number which is being computed. The others are just rough notes to “assist the memory”. It will only be these rough notes which will be liable to erasure.

Turing Machine Problems

* string of 0's and 1's contains an even number of 1's.
* string of a's and b's contains "aba" at some point
* string starts and ends with same letter.
* string is a palendrome
* add 1 to a binary number.
* return true if number of a's equals number of b's.
* test if two binary numbers separated by # are equal.
* write a TM to check if the number of 1's in the input is a power of 2.
* return true if number of a's equals number of b's equals number of c's.

Turing Machine Consequences

* can represent the complete description of a TM as a string of symbols.
* the universal turing machine U that takes in M and T and simulates M's computation on T. **why is this idea important?**
* **UNIVERSALITY IS THE HEART OF COMPUTER SCIENCE**
* halting problem. is there a maching H that takes in M and T and returns true if M(T) halts, and false if M(T) would enter an infinite loop.
* assume there is. can use this to create a machine G(M). if H(M, M) returns true, then loop forever. otherwise, return true. what happens if I now run G(G)?
* we have a contradiction. what does this prove?
* the halting problem for Turing machines is uncomputable: it is not possible to algorithmically decide whether a given Turing machine will ever halt
* computable numbers: a number is computable if there exists an algorithm to compute it to within some arbitrary range of precision. example: pi, e.
* countable number of algorithms, therefore a countable number of computable numbers.
* there are an uncountably infinite number of real numbers that are uncomputable!
* for example, the probability that a randomly selected turing machine will halt is a non-computable number.

Later Turing

* **Church-Turing thesis**, namely that any practical computing model has either the equivalent or a subset of the capabilities of a Turing machine.
* during **World War II**, Turing was a **pivotal player in breaking German U-Boat encryption system: Enigma**. (why was this good?)
* **designed one of the earliest electronic programmable digital computers** and **actually built** it.
* With the **Turing Test**, he made a significant and characteristically provocative contribution to the debate regarding synthetic consciousness: whether it will ever be possible to say that a machine is conscious and can think.
* homosexual. 1952: arrested and convicted. humiliating sentence.
* In 1954, he died of cyanide poisoning from a cyanide-laced apple he left half-eaten.
* The **Turing Award** was created in his honour.

Later

* Babbage – can omit
* Von Neumann – can omit

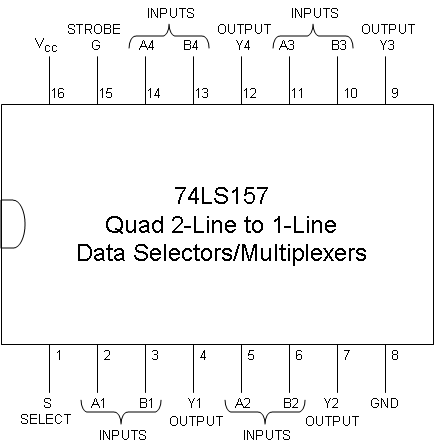
Charles Babbage

* 1791 – 1871
* British
* analytic engine in 1837
* mechanical, based on a weaving loom, used punch cards. never built.

# 5 CHUMP

## 5.0 Reference: CHUMP Chips

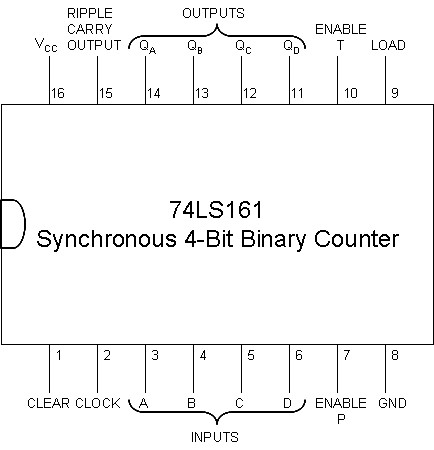
### 74LS157: Quad 2-Line to 1-Line Data Selectors/Multiplexers



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 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** |

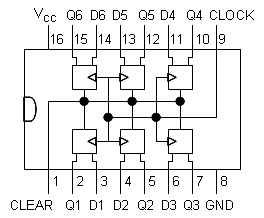
STROBE G should normally be wired low. A high level at the STROBE G input will set all outputs to low.

### 74LS161: Synchronous 4-Bit Binary Counter



* When LOAD is high, outputs will increment after each clock pulse (where QA 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.

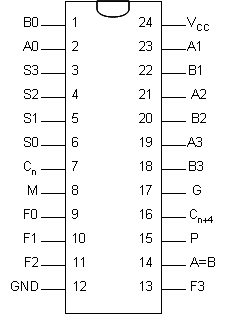
### 74LS174: Hex D-Type Flip-Flops with Clear



|  |  |  |  |
| --- | --- | --- | --- |
| Clear | Clock | D | **Q** |
| L | X | X | **L** |
| H | ↑ | H | **H** |
| H | ↑ | L | **L** |
| H | L | X | **Q0** |

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



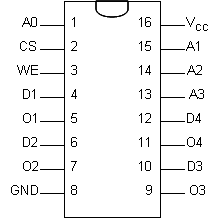
|  |  |
| --- | --- |
| Pin Names | Description |
| A0 – A3 | Operand Inputs |
| B0 – B3 | Operand Inputs |
| S0 – S3 | Function Select Inputs |
| M | Mode Control Input |
| Cn | Carry Input |
| F0 – F3 | Function Outputs |
| A=B | Comparator Output |
| G | Carry Generate Output |
| P | Carry Propagate Output |
| Cn+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) (Cn = 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 Cn = L adds 1 to output.

### 74S189: 64-Bit Random Access Memory

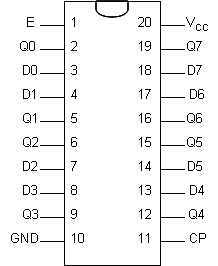


|  |  |
| --- | --- |
| Pin Names | Description |
| A0 – A3 | Address Inputs |
| CS | Chip Select Input (Active LOW) |
| WE | Write Enable Input (Active LOW) |
| D1 – D4 | Data Inputs |
| O1 – O4 | Inverted Data Outputs (Open Collector) |

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 |

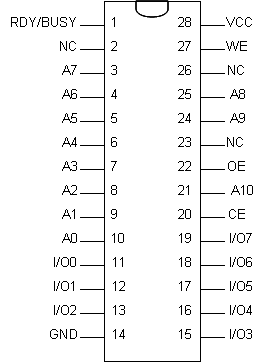
### 74LS377: 8-Bit Register



|  |  |
| --- | --- |
| 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** |

### 28C17: 16K (2K x 8) Parallel EEPROM



|  |  |
| --- | --- |
| Pin Name | Function |
| A0 – A10 | Addresses |
| CE | Chip Enable |
| OE | Output Enable |
| WE | Write Enable |
| I/O0 – I/O7 | Data Inputs/Outputs |
| RDY/BUSY | Ready/Busy Output |
| NC | No 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.

## 5.1 Lab: CHUMP

In this lab, you will build a CHUMP, a 4-bit minimal computer processor. 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!*

You will want to adopt a color convention for your wires. One such convention is to pick a different color for each "bus". 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 lab 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 ROM'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!*

This lab is organized in three parts, *each of which must be checked off by your teacher before you go on to the next part*. In the first part, you'll build the main datapath, with its many control inputs. In the second, you'll use a 4-bit opcode to control those many inputs. In the last, you'll complete the processor and run a program on it.

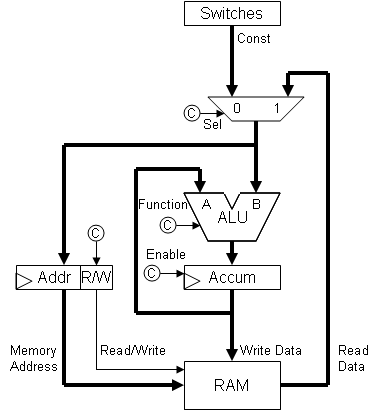
### The CHUMP Instruction Set

|  |  |
| --- | --- |
| Symbol | Meaning |
| const | constant embedded in instruction |
| addr | address flip-flops |
| mem | RAM memory |
| accum | accumulator register |
| pc | program counter |

|  |  |  |  |
| --- | --- | --- | --- |
| Instruction | OpCode | Summary | Description |
| LOAD const | 0000 | accum = const;  pc++; | Load constant into accumulator. |
| LOAD IT | 0001 | accum = mem[addr];  pc++; | Load read value into accumulator. |
| ADD const | 0010 | accum += const;  pc++; | Add constant to accumulator. |
| ADD IT | 0011 | accum += mem[addr];  pc++; | Add read value to accumulator. |
| SUBTRACT const | 0100 | accum -= const;  pc++; | Subtract constant from accumulator. |
| SUBTRACT IT | 0101 | accum -= mem[addr];  pc++; | Subtract read value from accumulator. |
| STORETO const | 0110 | mem[const] = accum;  pc++; | Store accumulator value to constant address. |
| STORETO IT | 0111 | mem[mem[addr]] = accum;  pc++; | Store accumulator value to read address. |
| READ const | 1000 | addr = const;  pc++; | Read from constant address. |
| READ IT | 1001 | addr = mem[addr];  pc++; | Read from read address. |
| GOTO const | 1010 | pc = const; | Jump to constant instruction number. |
| GOTO IT | 1011 | pc = mem[addr]; | Jump to read instruction number. |
| IFZERO const | 1100 | if (accum == 0)  pc = const;  else  pc++; | Jump to constant instruction number if accumulator is zero. |
| IFZERO IT | 1101 | if (accum == 0)  pc = mem[addr];  else  pc++; | Jump to read instruction number if accumulator is zero. |

### Part 1: The CHUMP Datapath

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.

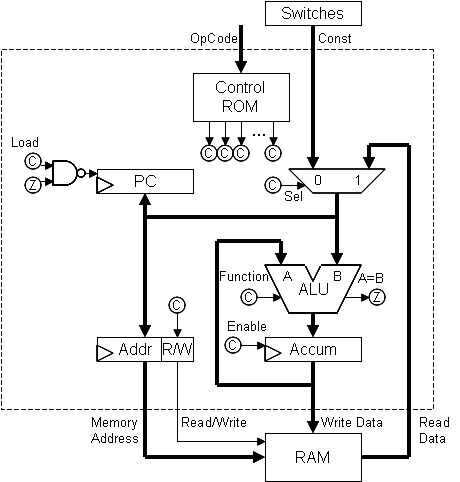


You will also need to fill in the following control table, and use it to test your CHUMP.

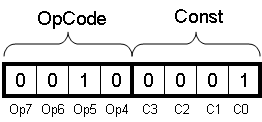
|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Instruction** | **Sel** | **(ALU)** | **S3** | **S2** | **S1** | **S0** | **M** | **Cn** | **Accum** | **R/W** |
| LOAD const | 0 | B | 1 | 0 | 1 | 0 | 1 | X | 0 | 1 |
| LOAD IT |  |  |  |  |  |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

### Part 2: CHUMP OpCodes

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.



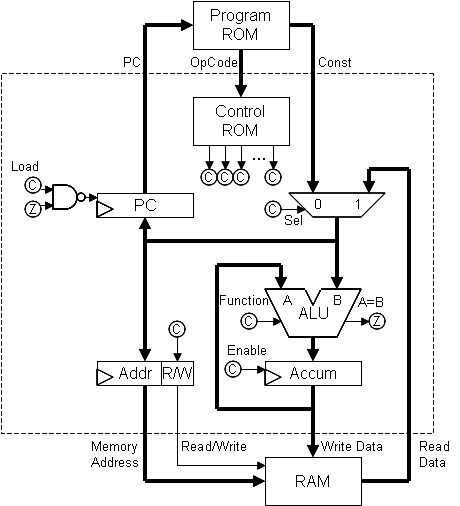
With a 4-bit OpCode, we could have as many as 16 distinct operations in our instruction set. Each 4-bit OpCode must be translated into the *ten* control bits required by our datapath (9 bits from part 1, plus a 10th to control the program counter). Go ahead and write down each instruction's 10 control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

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Inst** | **Op**  **7** | **Op**  **6** | **Op**  **5** | **Op**  **4** | **Sel** | **ALU** | **S**  **3** | **S**  **2** | **S**  **1** | **S**  **0** | **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 | 0 | 1 | 0 |  |  |  |  |  |  |  |  |  |  |  |
| GOTO IT | 1 | 0 | 1 | 1 |  |  |  |  |  |  |  |  |  |  |  |
| IFZERO const | 1 | 1 | 0 | 0 |  |  |  |  |  |  |  |  |  |  |  |
| IFZERO IT | 1 | 1 | 0 | 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.

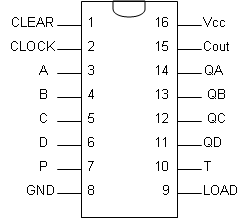
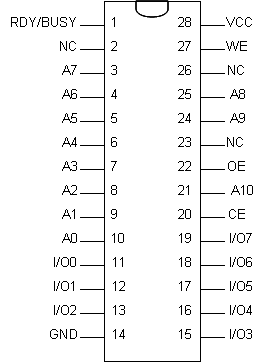
Part 3: Programming Your CHUMP

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.

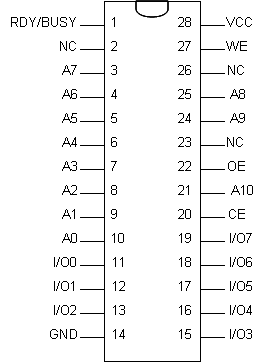
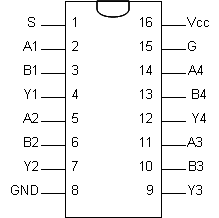


### Chip Diagrams

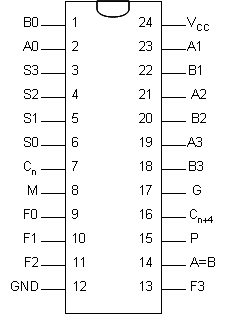
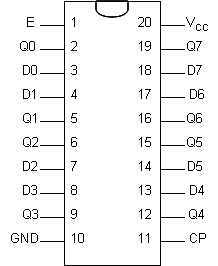
Mark all of your connections on these diagrams.

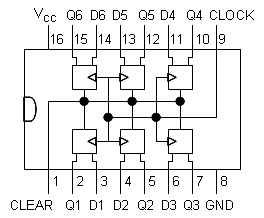
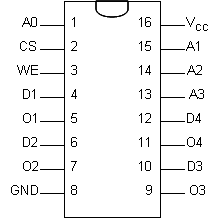
*74LS161: Program Counter 28C17: Program ROM*

*28C17: Control ROM 74LS157: Selector*

*74LS181: ALU 74LS377: Accumulator Register*

*74LS174: Address Flip-Flop 74S189: RAM*

# 6. Chumpanese

## 6.0 Chumpanese Guidelines

1. Every READ instruction must be followed by an IT instruction, and every IT instruction must be preceded by a READ instruction.

READ \_\_

\_\_\_\_ IT

2. A variable is simply a name for the memory address where its value is stored.

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

READ x LOAD IT

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

STORETO x

5. An if/else statement can be written as follows.

*<evaluate test>*

IFZERO zerocase

*<nonzero case>*

GOTO after

zerocase: *<zero case>*

after:

For example:  
 (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** 5 | addr=5; pc++ |
|  | 0001 | **0001** 0000 | **LOAD** IT | accum=[5]; pc++ |
|  | 0010 | **0100** 0011 | **SUBTRACT** 3 | accum-=3; pc++ |
|  | 0011 | **1100** 0111 | **IFZERO** 7 | accum=0?pc=7:pc++ |
| y = 1; | 0100 | **0000** 0001 | **LOAD** 1 | accum=1; pc++ |
|  | 0101 | **0110** 1010 | **STORETO** 10 | [10]=accum; pc++ |
|  | 0110 | **1010** 1001 | **GOTO** 9 | pc=9 |
| y = 2; | 0111 | **0000** 0010 | **LOAD** 2 | accum=2; pc++ |
|  | 1000 | **0110** 1010 | **STORETO** 10 | [10]=accum; pc++ |
|  | 1001 | **1010** 1001 | **GOTO** 9 | pc=9 //∞ loop! |

6. A while loop (with a != condition) can be written as follows.

loop: *<evaluate test>*

IFZERO after

*<loop body>*

GOTO loop

after:

For example:

while (n != 0) loop: READ n LOAD IT

n--; IFZERO after

READ n LOAD IT

SUBTRACT 1 STORETO n

GOTO loop

after:

7. To access an array value (a[2]) or a field (p1.y), first compute the address and store it to a temporary location. Reading from that address will require a READ IT instruction, while storing to it will require a STORETO IT instruction.

x = a[2]; READ a LOAD IT ADD 2 STORETO temp

READ temp READ IT LOAD IT

STORETO x

p1.y = 3; READ p1 LOAD IT ADD y STORETO temp

LOAD 3

READ temp STORETO IT

8. PUSH and POP are written as shorthand for sequences of instructions.(*not fully understood yet*)

PUSH = STORETO pushtemp ;must acquire address of top of

;stack first, so park the value to be

;pushed to memory

READ stack STORETO IT ;ACCUM ←address of top of stack

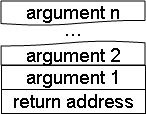
READ stack LOAD IT SUBTRACT 1 STORETO stack

READ pushtemp LOAD IT

POP = READ stack LOAD IT ADD 1 STORETO stack

READ stack READ IT LOAD IT

9. When a procedure is called, the top of the stack appears as follows. All of these values will be popped off the stack before the procedure returns.



A procedure can be written as follows.

procname: *<pop arguments in reverse order>*

POP STORETO continue

*<procedure body>*

*<load return value into accumulator>*

READ continue GOTO IT

For example:

public inc(n) inc: POP STORETO n

{ POP STORETO continue

return n + 1; READ n LOAD IT ADD 1

} READ continue GOTO IT

10. A procedure may be called as follows.

*<push any values needed after call>*

LOAD after PUSH

*<push arguments in order>*

GOTO procname

after: *<return value is in accumulator>*

*<pop saved values>*

For example:

x = inc(3); LOAD after PUSH

LOAD 3 PUSH

GOTO inc

after: STORETO x

## 6.1 Written Lab: Chumpanese

Translate each of the following to Chumpanese. (The number of blank lines corresponds to the suggested number of instructions.)

1. if (p == 3) 2. count = 0;

a = 4; while (count != 5)

else count = count + 1;

b = 5; count = 0;

c = 6;

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Consider the following class definition and variable declarations.

public class Person

{

public int age;

public int cats;

}

Person sue;

Person[] peeps;

Now translate each of the following to Chumpanese. Use instance variable names in your code, rather than making assumptions about offset values. Use the PUSH and POP macros when appropriate.

3. sue.age = 12; 4. sue.cats = sue.cats + 1;

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

5. peeps[3] = sue; 6. sue = peeps[index + 1];

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

7. int not(int x)

{ return 1 – x; }

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

8. x += not(y); 9. void setCats(Person p,

int num)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ { p.cats = num; }

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

## 6.2 Worksheet: Chumpanese

If we had sufficient time and memory for our Chumpanese programs, how powerful would this language be? We know that Java is universal (equivalent in power to the language of turing machines, since we can simulate any turing machine in Java). Is Chumpanese as powerful as Java, and therefore universal? To explore that question, we'll attempt to translate advanced Java constructs into Chumpanese.

## Review

Here's a simple Java program, which sums up the integers from 0 to n.

sum = 0;

while (n != 0)

{

sum += n;

n--;

}

Recall that we can translate this program into Chumpanese as follows.

LOAD 0

STORETO sum

loop: READ n

LOAD IT

IFZERO end

READ sum

ADD IT

STORETO sum

READ n

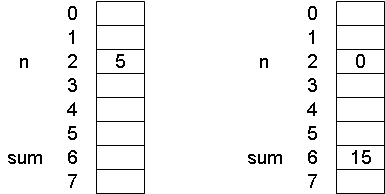
LOAD IT

SUBTRACT 1

STORETO n

GOTO loop

Recall that Chumpanese allows us to use names as shorthand for numeric line numbers and memory locations (RAM addresses). The name loop above is shorthand for "whatever line number is associated with the line labeled loop". If the program starts at line number 0, then loop is shorthand for the number 2. Likewise, the names sum and n are shorthand for the addresses of two memory locations. If we use memory address 2 for n and 6 for sum, then the following diagrams represent snapshots of RAM before and after running our program.



*Before After*

Finally, here's how our program would appear without the use of names as shorthand.

0: LOAD 0

1: STORETO 6

2: READ 2

3: LOAD IT

4: IFZERO 13

5: READ 6

6: ADD IT

7: STORETO 6

8: READ 2

9: LOAD IT

10: SUBTRACT 1

11: STORETO 2

12: GOTO 2

## Objects

The fundamental building block of object-oriented data structures is the object. Consider the following definition for the Point class.

public class Point

{

public int x;

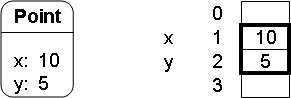
public int y;

public Point(int initX, int initY)

{ x = initX; y = initY; }

}

How can we represent such an object in RAM? Simply store the value of each variable in consecutive memory addresses, as shown here.



*Instance Diagram Representation in RAM*

But how can we represent a variable of type Point? In Java, variables come in two flavors—primitive types and reference types. So far, we have only translated Java programs that contain variables of the primitive type int. Handling the primitive type boolean is equally straightforward (we can use the value 0 to represent false and 1 to represent true), while handling the primitive type double is a bit more complicated. We have, however, entirely ignored reference types, such as Object, Integer, String, Point, int[], etc.

* A variable of a **primitive type** names a memory location that contains *a primitive value*.
* A variable of a **reference type**, also known as a **pointer**, names a memory location that contains *the address of another memory location*.

We saw earlier that the primitive int variables sum and n named memory locations that contained their integer values.

Suppose that we now define some reference type variables.

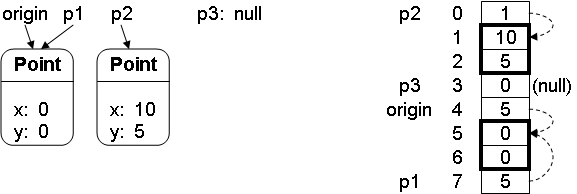
Point p1 = new Point(0, 0);

Point p2 = new Point(10, 5);

Point p3 = null;

Point origin = p1;

Since Point is a reference type, a variable of type Point will contain the address of the memory location where the corresponding Point object exists, as shown here. (Note that we've chosen to use 0 to represent the value null.)



For example, the value in p1 is 5, which tells us that the corresponding Point object appears in memory starting from address 5. The instance variables x and y can now be seen as shorthand names for the offsets from this starting address. In the example above, x would be shorthand for the offset 0, and y for the offset 1. In our Chumpanese code, however, *we won't make any assumptions about offset values*, and will stick to using instance variable names.

Whenever we work with pointers in Chumpanese, we'll first determine the memory address in question. At this point, the address we want will be in the accumulator, and we'll need it in the address flip-flops in order to read from or write to that address. Unfortunately, the only pathway from the accumulator to the address flip-flop is through memory itself, so we'll need to store the address to a temporary location. Later, we'll read from this temporary location, and use it as our next address with a READ IT command (to use the value in question) or a STORETO IT command (to change the value).

1. Determine the target memory address.

2. Store it to a temporary address.

3. (Possibly do some stuff.)

4. Read from the temporary address.

5. Use a READ IT or STORETO IT command.

Using these ideas, translate each of the following to Chumpanese.

p1.y = 2; val = p2.x;

### Linked Lists

Since linked lists consist only of objects, we should now be able to implement linked lists in Chumpanese. Suppose we define a linked list node as follows.

public class Node

{

public int value;

public Node next;

public Node(int initValue, Node initNext)

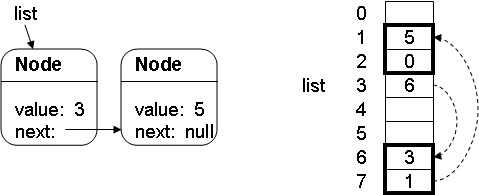
{ value = initValue; next = initNext; }

}

We now execute the following code.

Node list = new Node(3, new Node(5, null));

Here's how our linked list might look in memory.



Translate each of the following to Chumpanese:

list.next.value = 3;

int sum = 0;

Node curr = list;

while (curr != null)

{

sum += curr.value;

curr = curr.next;

}

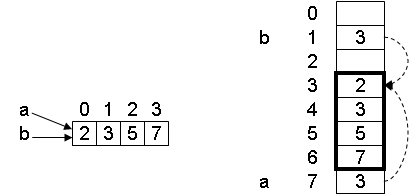
## Arrays of Primitives

Consider the following array declarations.

int[] a = {2, 3, 5, 7};

int[] b = a;

We can represent these arrays in memory using a sequence of consecutive memory locations.



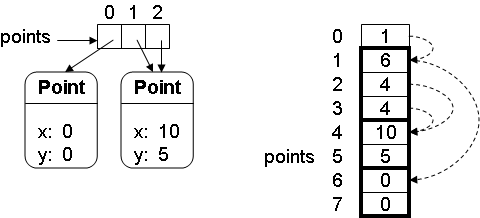
*Pictorial Representation Actual Representation in RAM*

Because array variables are really reference types, the array variable's RAM location contains the starting address of the array. An array index is then just an offset from the array's starting location. That's why array indices begin with 0! Try translating the following statements to Chumpanese.

a[2] = x; y = a[x]; a[i] = a[i + 1];

## Arrays of Objects

An array of objects can be represented as consecutive RAM locations containing the *addresses* of the objects, as shown here.

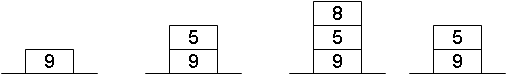


Try translating the following to Chumpanese.

a[1].y = 3; z = a[i].x;

## Stacks

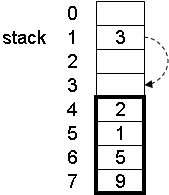
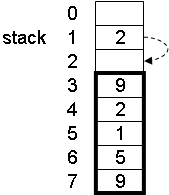
For reasons that will be clear soon, it will be very useful to represent a stack in our Chumpanese world. We would like to be able to PUSH a value from the accumulator onto the top of the stack, and POP a value from the top back into the accumulator, as in the following example. After executing these commands, the accumulator will contain the value 8.



LOAD 9 LOAD 5 LOAD 8 POP

PUSH PUSH PUSH

We can represent a stack as an expandable array, and keep track of a pointer to the top of the stack. By convention, we'll consider the top of the stack to be the beginning of the sequence (lowest memory address), and we’ll have the stack pointer stack refer to the first available space above the stack. The following diagrams show the value 9 being pushed onto the top of the stack (or popped off the stack, depending on which order you look at the pictures).

In Chumpanese, the macros PUSH and POP are shorthand for sequences of Chumpanese commands. Go ahead and write out these sequences.

PUSH POP

## A First Pass at Procedures

We'll pretend that Java has procedures, so that we don't need to worry about the distracting complications of calling methods instead. We'll represent a procedure as a *sequence of instructions beginning with a label*. Before calling the procedure, we'll leave its argument in the accumulator. When the procedure finishes, it will leave its result in the accumulator. Here's a simple "Java procedure" called double, and our first attempt to translate it into Chumpanese. (Yes, doubling a value in Chumpanese is a bit painful.)

int double(x) { return 2 \* x; }

double: STORETO x

READ x

LOAD IT

READ x

ADD IT

Now we'll translate a procedure foo that calls double.

int foo(n) { return double(n + 2) – 1; }

foo: ADD 2

GOTO double

after: SUBTRACT 1

This plan should bother you. Once the double code finishes executing, how will the processor know to return to the SUBTRACT 1 instruction in the foo code? We can't just add a GOTO after instruction to the end of the double code, *because double might be called by other procedures.*

## Returning from Procedures

Before calling a procedure, we'll store the line number to return to in a memory location we'll name continue. Then, at the end of each procedure we'll jump to the stored line number. (And, since we'll be jumping to a line number stored in memory, we can finally leverage the JUMP IT instruction!) Here's the revised version of our procedures.

double: STORETO x

READ x

LOAD IT

READ x

ADD IT

**READ continue**

**GOTO IT**

foo: STORETO n

**LOAD after-double**

**STORETO continue**

READ n

LOAD IT

ADD 2

GOTO double

**after-double:** SUBTRACT 1

Now we've correctly handled the call to double, but we haven't handled the call to foo correctly. Before calling foo, another procedure would have stored the return line number in continue, so foo must eventually jump to it. But if we just add READ continue and JUMP IT at the end of foo, we'll have a problem, since we've *already overwritten* the old value of continue that foo needed to jump back to. How can we solve this?

There are a couple other things we should now be concerned about. We don't yet have a way to pass multiple arguments to a procedure. In a way, the continue value we need is a second argument, so maybe we can solve both problems at once. Finally, we should also be worried that a call to a procedure could clobber the values of any variables we were using. The procedure we're calling might use the same variable names—especially if we're making a recursive call!

## The Procedure Stack

A stack is the solution to all our problems. Stacks are great for handling situations where we're in the middle of one problem when we get interrupted by another.

* We can use the stack to store our continue value.
* We can pass procedure arguments by placing them on the stack.
* We can store values on the stack that may get clobbered by a procedure call.

At the beginning and end of a program, the stack will be empty, and each PUSH in our code must have a corresponding POP. Here are our conventions for using the procedure stack. We'll use the name *continue* to refer to "the place to return to", even though the variable continue might not literally appear in our programs.

Before calling a procedure call

1. Push any values we'll need after the procedure call.

2. Push *continue*.

3. Push the arguments in order.

At the beginning of a procedure

Pop off the arguments in *reverse order*.

At the end of a procedure

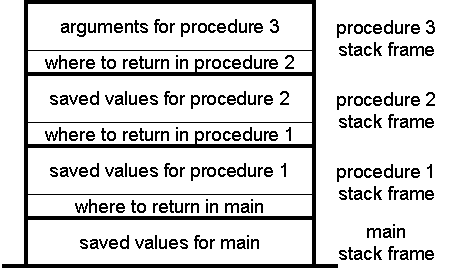
1. Pop off *continue*.

2. Go to *continue*.

After a procedure call

Pop off any saved values.

If we stick to these conventions, our procedure stack will typically look something like this. Here, "procedure 3" has just been called.



Here is the revised version of our code, using these conventions. For readability, we will always use the macro instructions PUSH and POP, but we should remember that these are merely shorthand for sequences of standard Chumpanese instructions.

double: **POP** //pop x off stack

STORETO x //find 2x

READ x

LOAD IT

READ x

ADD IT

STORETO result //save 2x

**POP** //pop/save continue

STORETO continue

READ result //put 2x into accum

LOAD IT

READ continue //return

GOTO IT

foo: **POP** //pop n off stack

ADD 2 //find n + 2

STORETO x //save n + 2

LOAD after-double //push after-double

**PUSH**

READ x //push n + 2

LOAD IT

**PUSH**

GOTO double //call double

after-double: SUBTRACT 1 //find double - 1

**POP** //pop/save continue

STORETO continue

READ continue //return

GOTO IT

Imagine how complicated this code would have been if we had needed to pass multiple arguments or to store any values for after the procedure call!

Here's a program called main that calls foo(3).

main: LOAD after-foo

PUSH

LOAD 3

PUSH

GOTO foo

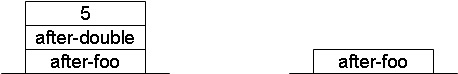
after-foo:

Let's see what values appear on the stack when we run main. At first, the stack is empty. By the time foo is called, though, the stack will contain the return point after-foo and the argument value 3.



*When main is reached When foo is reached*

When double is called, 3 has already been popped off the stack, and has been replaced with the return point after-double and the argument value 5. Once double has run its course, the 5 has been used to compute double's return value 10 (which now appears in the accumulator), and after-double has been used to return to that point.



*When double is reached When after-double is reached*

Finally, foo finds its result to be 9 and leaves it in the accumulator, and after-foo is popped off and used to return from foo.

## Recursion

Now that we know how to call procedures, we can implement a recursive procedure like multiply.

int multiply(int value, int times)

{

if (times == 0)

return 0;

return value + multiply(value, times – 1);

}

This procedure has two arguments. Since our stack convention tells us to push the arguments in order (value, then times), we'll get them back in *reverse order* when we pop them at the beginning of the procedure (times, then value). Also, notice that we'll need to use value after the recursive call. For this reason, we'll save value on the stack before the recursive call and pop it off when we return.

multiply: POP //pop/save times

STORETO times

POP //pop/save value

STORETO value

READ times //if (times == 0)

LOAD IT

IFZERO base-case

recur-case: SUBTRACT 1 //times - 1

STORETO times

READ value //push value

LOAD IT //(need it later)

PUSH

LOAD after //push after

PUSH

READ value //push value arg

LOAD IT

PUSH //push times arg

READ times

LOAD IT

PUSH

GOTO multiply //call multiply

after: STORETO result //save result

POP //pop value

READ result //add value

ADD IT

STORETO result //save result

POP //pop/save continue

STORETO continue

READ result //put result in acc

LOAD IT

READ continue //return

GOTO IT

base-case: POP //pop/save continue

STORETO continue

LOAD 0 //put 0 in accum

READ continue //return

GOTO IT

The following main program calls multiply(5, 2).

main: LOAD end

PUSH

LOAD 5

PUSH

LOAD 2

PUSH

GOTO multiply

end:

Let's see what happens when main is run. The following table shows the order in which the instruction labels are reached, along with the stack and accumulator contents at those times.

|  |  |  |
| --- | --- | --- |
| **Label Reached** | **Stack Contents** | **Accumulator Contents** |
| main |  |  |
| multiply |  |  |
| recur-case |  | 2 |
| multiply |  |  |
| recur-case |  | 1 |
| multiply |  |  |
| base-case |  | 0 |
| after-mult |  | 0 |
| after-mult |  | 5 |
| end |  | 10 |

It should now be clear that an infinite recursion would result in a stack overflow (a condition that a fancier version of PUSH could detect).

# Email Notes from D. Feinberg

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 mis-wired.