Extending Toddler

-1xx  LOADX  xx  Loads the value from address xx into XR
-2xx  STOREX  xx  Stores the value from XR into address xx
-3xx  LOAD  xx (XR)  Loads AC with the contents of xx + XR
-4xx  STORE  xx (XR)  Stores AC into address xx + XR
-500  RETURN  Returns from a function
-5xx  CALL  xx  Call the function at address xx
-6xx  PUSH  xx  Push the contents of xx on the stack
-7xx  POP  xx  Pops the top element on the stack into xx
-8xx  INCHAR  xx  Reads a character code into address xx
-9xx  OUTCHAR  xx  Prints the character code in address xx

Chris Gregg, based on slides by Eric Roberts
CS 208E
October 25, 2018
The Concept of a Stack

• A **stack** is a data structure in which the elements are accessible only in a *last-in/first-out* order. The operations on a stack are **push**, which adds a value to the top of the stack, and **pop**, which removes and returns the top value.

• One of the most common metaphors for the stack concept is a spring-loaded storage tray for dishes. Adding a new dish to the stack pushes any previous dishes downward. Taking the top dish away allows the dishes to pop back up.

• Stacks are important in von Neumann machines because function calls obey a last-in/first-out discipline.
The Toddler System Stack

• Like all modern hardware, the Toddler machine implements a stack in hardware to simplify dividing programs up into independent functions.

• The Toddler stack lives at the highest addresses in memory, so the bottom of a stack is at address 99, and the stack grows toward lower memory addresses.

• The address of the element at the top of the stack is stored in the register SP. If the SP is 00, that means the stack is empty.

• Pushing a value on the stack corresponds to subtracting one from the SP and then storing a value in the resulting address.

• Popping the top value from the stack reverses the process by taking the current contents of the word addressed by SP and then adding one to SP.
Functions and Stacks

• The **CALL** instruction pushes the current value of the **PC** (which has already been incremented to refer to the next instruction) on the stack. This value is called the *return address*.

• The **RETURN** instruction pops the top value on the stack into the **PC**, which has the effect of returning to the point just after the **CALL** instruction.
## The Extended Instruction Set

<table>
<thead>
<tr>
<th>Number</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1xx</td>
<td>LOADX xx</td>
<td>Loads the value from address xx into XR</td>
</tr>
<tr>
<td>-2xx</td>
<td>STOREX xx</td>
<td>Stores the value from XR into address xx</td>
</tr>
<tr>
<td>-3xx</td>
<td>LOAD xx (XR)</td>
<td>Loads AC with the contents of xx + XR</td>
</tr>
<tr>
<td>-4xx</td>
<td>STORE xx (XR)</td>
<td>Stores AC into address xx + XR</td>
</tr>
<tr>
<td>-500</td>
<td>RETURN</td>
<td>Returns from a function</td>
</tr>
<tr>
<td>-5xx</td>
<td>CALL xx</td>
<td>Call the function at address xx</td>
</tr>
<tr>
<td>-6xx</td>
<td>PUSH xx</td>
<td>Push the contents of xx on the stack</td>
</tr>
<tr>
<td>-7xx</td>
<td>POP xx</td>
<td>Pops the top element on the stack into xx</td>
</tr>
<tr>
<td>-8xx</td>
<td>INCHAR xx</td>
<td>Reads a character code into address xx</td>
</tr>
<tr>
<td>-9xx</td>
<td>OUTCHAR xx</td>
<td>Prints the character code in address xx</td>
</tr>
</tbody>
</table>
Exercise: Multiply as a Function

• Rewrite the `Multiply.td` program so that it defines a function called `mult` that takes values in the variables `n1` and `n2` and returns its answer in a variable called `result`.

• Use that function to write a program called `Factorial.td` that computes the factorial of an integer. The largest factorial that fits in three digits is 6!, so a sample run might look like this:
The Extended Instruction Set

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>LOADX xx</code></td>
<td>Loads the value from address <code>xx</code> into <code>XR</code></td>
</tr>
<tr>
<td><code>STOREX xx</code></td>
<td>Stores the value from <code>XR</code> into address <code>xx</code></td>
</tr>
<tr>
<td><code>LOAD xx (XR)</code></td>
<td>Loads <code>AC</code> with the contents of <code>xx + XR</code></td>
</tr>
<tr>
<td><code>STORE xx (XR)</code></td>
<td>Stores <code>AC</code> into address <code>xx + XR</code></td>
</tr>
<tr>
<td><code>RETURN</code></td>
<td>Returns from a function</td>
</tr>
<tr>
<td><code>CALL xx</code></td>
<td>Call the function at address <code>xx</code></td>
</tr>
<tr>
<td><code>PUSH xx</code></td>
<td>Push the contents of <code>xx</code> on the stack</td>
</tr>
<tr>
<td><code>POP xx</code></td>
<td>Pops the top element on the stack into <code>xx</code></td>
</tr>
<tr>
<td><code>INCHAR xx</code></td>
<td>Reads a character code into address <code>xx</code></td>
</tr>
<tr>
<td><code>OUTCHAR xx</code></td>
<td>Prints the character code in address <code>xx</code></td>
</tr>
</tbody>
</table>
Class Example: Hello, World

• The **INCHAR** and **OUTCHAR** instructions are similar to **INPUT** and **OUTPUT** except that they read and write the numeric representation of a single character.

• The rest of this lecture develops three implementations of a program that prints the string “hello, world” on the console.
Hello World: The Brute Force Version

/*
 * File: HelloWorld1.td
 * ----------------------
 * Writes out "hello, world" character by character.
 */

start: OUTCHAR #104    /* The code for the character 'h' */
      OUTCHAR #101    /* 'e' */
      OUTCHAR #108    /* 'l' */
      OUTCHAR #108    /* 'l' */
      OUTCHAR #111    /* 'o' */
      OUTCHAR #44     /* ',' */
      OUTCHAR #32     /* ' ' */
      OUTCHAR #119    /* 'w' */
      OUTCHAR #111    /* 'o' */
      OUTCHAR #114    /* 'r' */
      OUTCHAR #108    /* 'l' */
      OUTCHAR #100    /* 'd' */
      OUTCHAR #10     /* The newline character ('
') */
HALT
Self-Modifying Code

• One of the defining features of the von Neumann architecture is that instructions and data are stored in the same memory. That fact makes it possible for programs to modify their own instructions by treating them just like any other numeric data.

• The HelloWorld2.td program uses this technique to create an instruction that prints a character from the address that is the start of the string "hello, world" plus the value of the index i. It then stores that instruction in the program and executes it.

• Programs that change their own instructions are said to be self-modifying. In early machines, this strategy was often the only way to accomplish certain operations. Today, it is generally seen as a dangerous programming practice.
Hello World: Self-Modifying Code

/*
 * File: HelloWorld2.td
 * -------------------
 * Writes out "hello, world" using self-modifying code.
 */

start:  LOAD #msg  /* Load the address of the string */
       CALL strout /* Call the function to output a string */
       HALT      /* And halt */

strout: STORE addr  /* Store current address */
       LOAD ldins /* Load a word with a LOAD 0 instruction */
       ADD addr  /* Add the address offset */
       STORE patch /* Store the LOAD in the next word */

patch:  0  /* This will get filled in */
       JUMPZ ret /* A zero character marks the end */
       STORE ch  /* Store the character */
       OUTCHAR ch /* Write it out */
       LOAD addr /* Get the current address */
       ADD #1 /* Move to the next one */
       JUMP strout /* And go back for more */

ret:    RETURN /* Return from the strout function */

msg:    "hello, world"
ldins:  LOAD 0
addr:   0
ch:     0
The Internet Worm

‘Virus’ in Military Computers Disrupts Systems Nationwide

By JOHN MARKOFF

As a result of the virus, the National Security Agency, the Defense Department, and the intelligence community have been hit hard. The virus has been able to spread through the computer network, making hundreds of copies of itself in each machine it reached, effectively spreading viruses through thousands of military, corporate, and university computers around the nation and preventing them from doing additional work.

By late yesterday afternoon, computer experts were calling the virus the largest virus ever on the nation’s computers.

“The problem is that the virus is a relatively benign software program that can spread very fast,” said Dr. John F. Cramer, director of the NSA’s Information Systems Research Laboratory. “The virus is a self-contained program that can replicate itself and spread to other computers.

The virus can copy itself from one computer to another, spreading quickly through the network. The virus can also disrupt the computer’s operating system, causing problems such as lost data or system crashes.

The virus spread through the network and caused widespread disruption. The NSA reported that the virus caused “unprecedented” problems, leading to a loss of communication and data.

The virus has been a significant threat to the military and intelligence communities, and efforts are underway to contain and remove it. The US Department of Defense has set up a task force to combat the virus, and the NSA has launched an investigation to determine the source of the virus.

The virus has also had significant implications for the business community, as it has caused significant damage to corporate networks and data centers. The virus has been found in multiple industries, including finance, healthcare, and retail.

The virus has also raised concerns about the security of the nation’s critical infrastructure. The virus has the potential to cause widespread damage to the nation’s power grid, transportation systems, and other vital services.

The virus has also raised concerns about the security of the nation’s critical infrastructure. The virus has the potential to cause widespread damage to the nation’s power grid, transportation systems, and other vital services.
How the Morris Worm Worked

If the user, however, enters a name string that overflows the buffer, the bytes in that name will overwrite the data on the stack.

Now when the function returns, it will jump into the code written as part of the name, thereby executing the worm’s instructions.
## The Extended Instruction Set

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(-1_{xx}) LOADX (xx)</td>
<td>Loads the value from address (xx) into XR</td>
</tr>
<tr>
<td>(-2_{xx}) STOREX (xx)</td>
<td>Stores the value from XR into address (xx)</td>
</tr>
<tr>
<td>(-3_{xx}) LOAD (xx) (XR)</td>
<td>Loads AC with the contents of (xx + XR)</td>
</tr>
<tr>
<td>(-4_{xx}) STORE (xx) (XR)</td>
<td>Stores AC into address (xx + XR)</td>
</tr>
<tr>
<td>(-500) RETURN</td>
<td>Returns from a function</td>
</tr>
<tr>
<td>(-5_{xx}) CALL (xx)</td>
<td>Call the function at address (xx)</td>
</tr>
<tr>
<td>(-6_{xx}) PUSH (xx)</td>
<td>Push the contents of (xx) on the stack</td>
</tr>
<tr>
<td>(-7_{xx}) POP (xx)</td>
<td>Pops the top element on the stack into (xx)</td>
</tr>
<tr>
<td>(-8_{xx}) INCHAR (xx)</td>
<td>Reads a character code into address (xx)</td>
</tr>
<tr>
<td>(-9_{xx}) OUTCHAR (xx)</td>
<td>Prints the character code in address (xx)</td>
</tr>
</tbody>
</table>
Index Registers

• The `HelloWorld3.td` program avoids the self-modifying strategy by using the Toddler machine’s `index register` (XR), which automatically adds the contents of the index register to the address given in a `LOAD` or `STORE` instruction.

• The `LOADX` and `STOREX` instructions load and store the contents of the XR itself. Supplying the (XR) suffix on a `LOAD` or `STORE` instruction changes what memory address is referenced.
Hello World: Using the Index Register

/*
  * File: HelloWorld3.td
  * ------------------
  * Writes out "hello, world" using the index register.
  */

start:  LOAD #msg /* Load the address of the string */
        CALL strout /* Call the function to output a string */
        HALT /* And halt */

strout: STORE addr /* Store current address */
        LOADX addr /* Load that address into the XR */
        LOADX 0(XR) /* Load the value at that address */
        JUMPZ ret /* A zero character marks the end */
        STORE ch /* Store the character */
        OUTCHAR ch /* Write it out */
        LOAD addr /* Get the current address */
        ADD #1 /* Move to the next one */
        JUMP strout /* And go back for more */

ret:    RETURN /* Return from the strout function */

msg:    "hello, world"
addr:   0
ch:     0
The End