Extending Toddler

Chris Gregg and Eric Roberts  
CS 208E  
October 20, 2016

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.

The Extended Instruction Set

- LOADX xx Loads the value from address xx into XR
- STOREX xx Stores the value from XR into address xx
- LOAD xx (XR) Loads AC with the contents of xx + XR
- STORE xx (XR) Stores AC into address xx + XR
- RETURN Returns from a function
- PUSH xx Push the contents of xx on the stack
- POP xx Pops the top element on the stack into xx
- INCHAR xx Reads a character code into address xx
- OUTCHAR xx Prints the character code in address xx

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.

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:
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

```
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

```
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 */
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 */
```

The Internet Worm

Robert Morris Jr.
How the Morris Worm Worked

Storage for local variables in Unix is provided by a stack, which grows toward low memory addresses as functions are called.

The *fingerd* code allocates a stack buffer to hold the user name, which might be declared like this:

```c
cchar buffer[20];
```

If the string supplied is too long, it will overwrite the contents of the stack and allow the worm to execute the inserted code.

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

```c
/* 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 */
LOAD addr /* Load that address into the XR */
JMPF zer /* 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
```