University of Illinois at Urbana-Champaign
Dept. of Electrical and Computer Engineering

ECE 220: Computer Systems & Programming

The Stack Abstraction

Conventions Provide Implicit Information

What does this mean: **1 + 2 × 3** ?

It could mean **(1 + 2) × 3 = 9** .

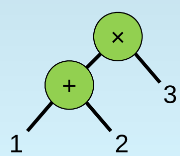
Or it could mean **1 + (2 × 3) = 7** .

Most (all?) cultures on Earth
- choose this one
- by **convention**.

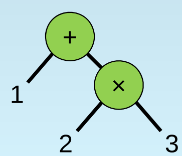
Arithmetic with Trees is Unambiguous

We can
- **eliminate ambiguity**
- **by using trees**.

(1 + 2) × 3



1 + (2 × 3)



Why Not Always Use Trees?

Since you're in ECE,
- I've asked your Math professors
- to let you use trees
- for all future homework.

Trees are painful for humans!

Sound good? Here's some practice…

Write *F(x,y)* and the partial derivatives of *F(x,y)* in *x* and *y*…**using trees**:

$$F(x,y) = \frac{1}{2}e^{-a(x^2+y^2)} - \cos\left(20x + \frac{\pi}{4}\right)$$

Other Notations are Also Unambiguous

Our usual notation (“1 + 2”)
- is called **infix** because
- **operators appear in between operands.**

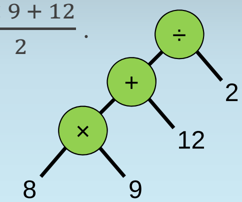
Postfix (and prefix) notation
- **is not ambiguous,**
- So it does not require parentheses!

For 20+ years, all HP engineering calculators used postfix (“reverse Polish”)…ask your parents.

Postfix Notation is a Programming Language!

For example, we write $\frac{8 \times 9 + 12}{2}$.

As a tree, we draw…



In postfix, we write

8 9 × 12 + 2 ÷

This version (postfix) is a program!

Let’s Run the Program…

Our program: **8 9 × 12 + 2 ÷**

Execute the “program” using a stack of paper:
- For a number,
  1. write number on a sheet of paper, and
  2. place it on top of the stack.
- For an operator,
  1. grab the top two sheets from the stack,
  2. perform the operation,
  3. write result on a sheet of paper, and
  4. place it on top of the stack.

R6 Points to the Top of Our Stack in LC-3 Memory

To compute our postfix program, we **used a stack of paper**.

Can we use computer memory instead?

Do you remember the idea of
- putting subroutine inputs/outputs
- into memory, then
- using a register
- to point to those memory locations?

For LC-3, use R6 to point to the top of our stack.*

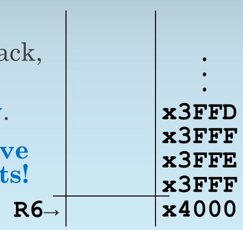
*A convention. Most ISAs have a register called the stack pointer.

When R6 Points to Base of Stack, Stack is Empty

Initially,
- R6 points to "base" of stack,
- let's say address x4000,
- and the **stack is empty**.

What is in memory above the top of the stack? Bits!
Hint: not "air," nor "blanks."

By convention, **those bits are NOT on the stack**.



To "Execute" a Number Instruction, Push Onto Stack

Let's run our program again:

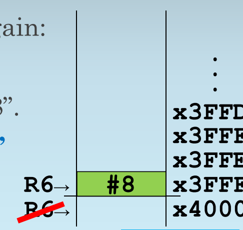
8 9 × 12 + 2 ÷

The first instruction is "8".

How can we put an "8" on the stack?

; Assume 8 in R0.
ADD R6,R6,#-1 ; make space first!
STR R0,R6,#0 ; then store the 8

called a "push"



Pushing R0 Always Uses the Same Two Instructions

Continue executing!

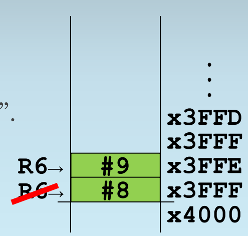
8 9 × 12 + 2 ÷

The next instruction is "9".

How can we put a "9" on the stack?

; (Put 9 in R0 here.)
ADD R6,R6,#-1 ; make space first!
STR R0,R6,#0 ; then store the 9

same two inst.!

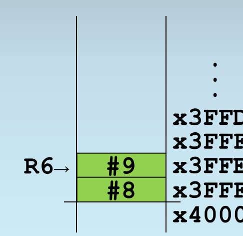


The Next Instruction is Multiply

What about multiply?

8 9 × 12 + 2 ÷

Assume that someone has written a multiply routine:
- subroutine MULT
- R0, R1 input operands
- R0 output (R0 ← R0 × R1)



Example of a MULT Subroutine

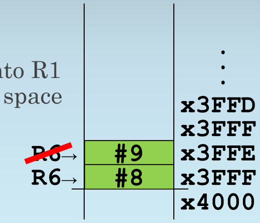
```
MULT
AND R2,R2,#0
ADD R1,R1,#0
BRz MULTDONE
MULTLOOP
ADD R2,R2,R0
ADD R1,R1,#-1
BRnp MULTLOOP
MULTDONE
ADD R0,R2,#0
RET
```

What is the call interface for this subroutine?

Inputs: R0, R1
Output: R0 ← R0 × R1
Caller-saved:
R1, R2, R7
Callee-saved:
R3, R4, R5, R6

To Multiply: Pop Twice, Multiply, Push Product

```
STACKMULT

LDR R1,R6,#0     ; pop 9 into R1
ADD R6,R6,#1     ; remove space
```

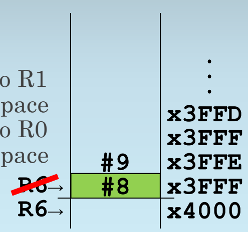


Is the "9" still in memory?

Probably, but it's NOT on the stack.

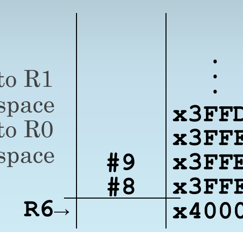
To Multiply: Pop Twice, Multiply, Push Product

```
STACKMULT

LDR R1,R6,#0     ; pop 9 into R1
ADD R6,R6,#1     ; remove space
LDR R0,R6,#0     ; pop 8 into R0
ADD R6,R6,#1     ; remove space
```



To Multiply: Pop Twice, Multiply, Push Product

```
STACKMULT

LDR R1,R6,#0     ; pop 9 into R1
ADD R6,R6,#1     ; remove space
LDR R0,R6,#0     ; pop 8 into R0
ADD R6,R6,#1     ; remove space
JSR MULT         ; R0 is 72
```



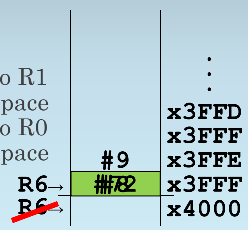
We're ready to call MULT!

Note that the stack is empty.

To Multiply: Pop Twice, Multiply, Push Product

STACKMULT

```
LDR R1,R6,#0     ; pop 9 into R1
ADD R6,R6,#1     ; remove space
LDR R0,R6,#0     ; pop 8 into R0
ADD R6,R6,#1     ; remove space
JSR MULT         ; R0 is 72
ADD R6,R6,#-1    ; push R0
STR R0,R6,#0
```



Use the same instructions as before!

That's it!

Subroutine Can Mean More than Just Adding RET

STACKMULT

```
LDR R1,R6,#0     ; pop 9 into R1
ADD R6,R6,#1     ; remove space
LDR R0,R6,#0     ; pop 8 into R0
ADD R6,R6,#1     ; remove space
JSR MULT         ; R0 is 72
ADD R6,R6,#-1    ; push R0
STR R0,R6,#0

RET
```

But what if we want a subroutine?

Good enough?

NO!

A Subroutine that Uses JSR or TRAP Must Protect R7

STACKMULT

```
 ; R7 has the return address here.
LDR R1,R6,#0     ; pop 9 into R1
ADD R6,R6,#1     ; remove space
LDR R0,R6,#0     ; pop 8 into R0
ADD R6,R6,#1     ; remove space
JSR MULT         ; R0 is 72
ADD R6,R6,#-1    ; push R0
STR R0,R6,#0

RET
```

Where does R7 point after JSR?

Here.

So RET creates a loop…

Add a Space with a Label, then Save and Restore R7

STACKMULT

```
ST R7,SM_R7      ; save R7
LDR R1,R6,#0     ; pop 9 into R1
ADD R6,R6,#1     ; remove space
LDR R0,R6,#0     ; pop 8 into R0
ADD R6,R6,#1     ; remove space
JSR MULT         ; R0 is 72
ADD R6,R6,#-1    ; push R0
STR R0,R6,#0
LD R7,SM_R7      ; restore R7
RET
SM_R7 .BLKW #1   ; space for R7
```

Now the subroutine is complete.

Review: the Stack Abstraction

Stack in memory similar to stack on a desk.

Operations include:
- **PUSH—put something on top of the stack**
- **POP—take the top thing off of the stack**

A stack
- provides last-in, first-out (LIFO) semantics:*
- first thing popped is the last thing pushed

*As opposed to first-in, first-out (FIFO) semantics, as with the queue that we used with BFS.

Review: the Stack Abstraction in LC-3

In LC-3,
- we use R6 as a stack pointer, and
- PUSH/POP require two instructions each

Most ISAs
- **have a stack pointer register and**
- **include PUSH/POP instructions.**

The Stack at This Level is Not Checked

P&P talk about overflow/underflow checks.

That's fine when we reach C.

High-level languages (such as C) rely heavily on the stack provided by the ISA.

The stack provided by the ISA
- **is typically unchecked**,
- as checking overhead is too high, so
- don't make mistakes.

What Really Happens with Overflow/Underflow?

If a **stack overflows**…
- in LC-3/embedded processor/inside OS,* causes **silent data corruption**;
- in desktop/laptop/phone application, hardware detects, and OS causes **program to crash**.

If a stack underflows…
- silent data corruption is likely to happen first, and
- program may crash.

*For example, inside your OS in ECE391.

What is a Think-Pair-Share?

A group exercise in lecture, not unlike discussion sections in ECE120.

The process:

1. I give you a problem.
2. You form groups of 3-4 people.
3. Talk about ways to solve the problem.
4. Once enough of the groups have finished, one group volunteers to share their answer.
5. We go over the group's answer together.

The Task: a Factorial Subroutine

Write subroutine FACTORIAL

- to compute output R0
- as the factorial of input R0.
- In other words, R0 ← 1 × 2 × … × R0.

Assumptions and rules…

- Assume that input R0 is at least 1.
- Assume that R6 points to a valid stack.
- Write your subroutine in LC-3 assembly language.
- Use the STACKMULT subroutine to calculate the answer.
- Clearly define the calling interface.

A Task on Your Own: 16-bit Palindrome Check

What's a palindrome?

- Same spelling backwards as forwards.
- Examples include "Otto" and "Hannah."

Your task:

- Check whether R0 is a palindrome.
- Example: 0111 1011 1101 1110.
- Return R0=1 if yes, R0=0 if no.

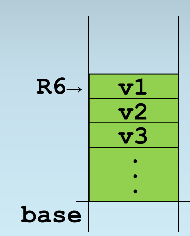
See sample solution on the web page.

Assumptions and rules…

- Assume that R6 points to a valid stack (use it).
- Write your code in LC-3 assembly language.

We Can Use Known Values on the Stack Directly

In practice, **we need not strictly obey the rules** of the stack abstraction.



Consider the following task:

- sum three non-negative values from top of the stack,
- pop all three values, and
- return the sum in R0.

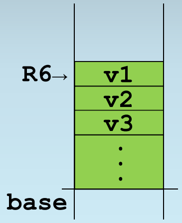
Let's assume that only R0 should change.

Let's Start by Saving R1 and Reading v1

```
SUM_OF_3
ST R1,SAVE_R1    ; save R1
LDR R0,R6,#0     ; R0 ← v1
```

So far, so good?

But we're not going to pop v1…




```
SAVE_R1 .BLKW #1
```

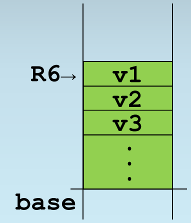
Make a space down here.

Load v2 Using LDR from M[R6 + 1]

```
SUM_OF_3
ST R1,SAVE_R1    ; save R1
LDR R0,R6,#0     ; R0 ← v1
LDR R1,R6,#1     ; R1 ← v2
ADD R0,R0,R1     ; R0 ← v1 + v2
```

Read v2 before popping v1.

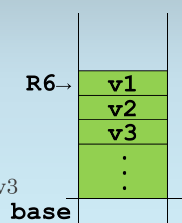
And find the sum…




```
SAVE_R1 .BLKW #1
```


Do the Same for v3 (with Offset 2)

```
SUM_OF_3
ST R1,SAVE_R1    ; save R1
LDR R0,R6,#0     ; R0 ← v1
LDR R1,R6,#1     ; R1 ← v2
ADD R0,R0,R1     ; R0 ← v1 + v2
LDR R1,R6,#2     ; R1 ← v3
ADD R0,R0,R1     ; R0 ← v1 + v2 + v3
```



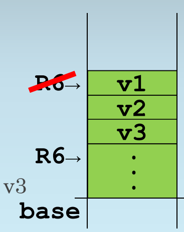
Now read v3.

And find the sum…

```
SAVE_R1 .BLKW #1
```


Pop All Three Values at Once

```
SUM_OF_3
ST R1,SAVE_R1    ; save R1
LDR R0,R6,#0     ; R0 ← v1
LDR R1,R6,#1     ; R1 ← v2
ADD R0,R0,R1     ; R0 ← v1 + v2
LDR R1,R6,#2     ; R1 ← v3
ADD R0,R0,R1     ; R0 ← v1 + v2 + v3
ADD R6,R6,#3     ; pop all three
```

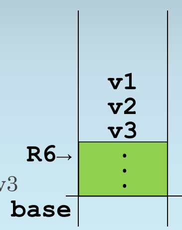


Done with the values: pop all three!

```
SAVE_R1 .BLKW #1
```


Finish by Restoring R1 and Returning

```
SUM_OF_3
ST R1,SAVE_R1      ; save R1
LDR R0,R6,#0       ; R0 ← v1
LDR R1,R6,#1       ; R1 ← v2
ADD R0,R0,R1       ; R0 ← v1 + v2
LDR R1,R6,#2       ; R1 ← v3
ADD R0,R0,R1       ; R0 ← v1 + v2 + v3
ADD R6,R6,#3       ; pop all three
LD R1,SAVE_R1      ; restore R1
RET
SAVE_R1 .BLKW #1
```



Restore R1 and return.

Breaking the Abstraction Can Be Done Safely

To use SUM_OF_3,
- push three values, call SUM_OF_3, and use the result in R0.
- Or allocate three locations with one ADD, write in three values, then call …

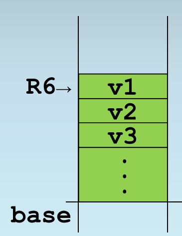
We can **safely use**
- **any data on the stack**
- **if we know that it's there**.

Can We Generalize SUM_OF_3 to SUM_OF_N?

The picture to the right shows
- an **array of three integers**
- on top of the stack.

What if we want to generalize?

Can we write a subroutine
- that **adds a variable number of non-negative numbers**
- **from an array on top of the stack**?

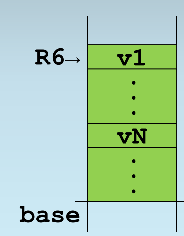


Can We Generalize SUM_OF_3 to SUM_OF_N?

Can we write a subroutine that adds N non-negative numbers from the top of the stack?

Yes!

But **the subroutine must know the value of N**.

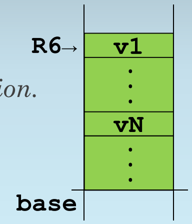


How Can the Subroutine Be Given N?

How can the caller tell the subroutine the value of N?

Hint: this is NOT a trick question. Give the easy answers first!

1. **Use a fixed value**, such as 3.
2. **Pass N in a register**, say R2.



The Answers Will Be Useful in Other Contexts

This question occurs in many contexts:

- determining array length
- passing variable numbers of arguments, and
- using network connections in applications.

Be sure that you understand the options!

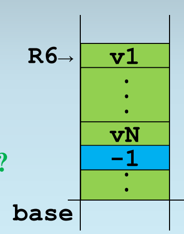
Another Solution: the ASCII String Approach

How can the caller tell the subroutine the value of N?

1. Use a fixed value, such as 3.
2. Pass N in a register, say R2.

How do ASCII strings work?

3. **End the list with a non-data sentinel (such as -1).***

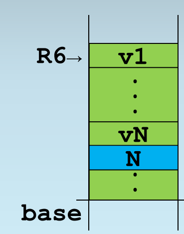


*Now you know why we assumed "non-negative."

Does Putting N at the End of the Array Work?

What if we put N at the end of the array?

Does such an approach work?



Does Putting N at the End of the Array Work?

Given the stack shown here, **what should the subroutine return?**

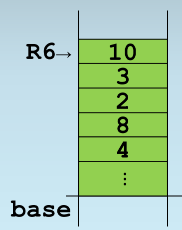
13? (N=2)

23? (N=4)

Something else? (Is N shown?)

The answer is ambiguous!

(Such an approach is not acceptable.)



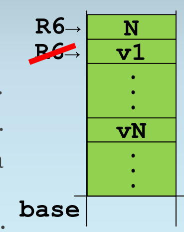
One Other Solution is Possible

How can the caller tell the subroutine the value of N?

1. Use a fixed value, such as 3.
2. Pass N in a register, say R2.
3. End the list with a non-data sentinel (such as -1).

But there is one more answer…

4. **Put N on top of the stack (always in a known position: M[R6]).**



A Stack for MP3

In MP3,
- you will use a stack
- to implement a depth-first search (DFS).

Given
- a list of extra events,
- each with several options for hour slot,
- you must try to find a combination
- that works without schedule conflicts.

A Stack Frame Holds All Information for a Subroutine

Imagine that you are using
- an ISA with few/no registers, so
- you must use the stack to manage subroutine calls.

Let's **define a block of data**
- called a **stack frame** (or **activation record**)
- that **holds all** of the **information**
- needed **for one subroutine**.

A Stack Frame Holds All Information for a Subroutine

What needs to be in a stack frame?

Local variables

Address of caller's stack frame

Return address (R7 in LC-3)

Outputs (return value)

Inputs (parameters, arguments)

these form the **linkage** (Address of caller's stack frame, Return address, Outputs)

You'll grow quite tired of these by March.