EXERCISE #10

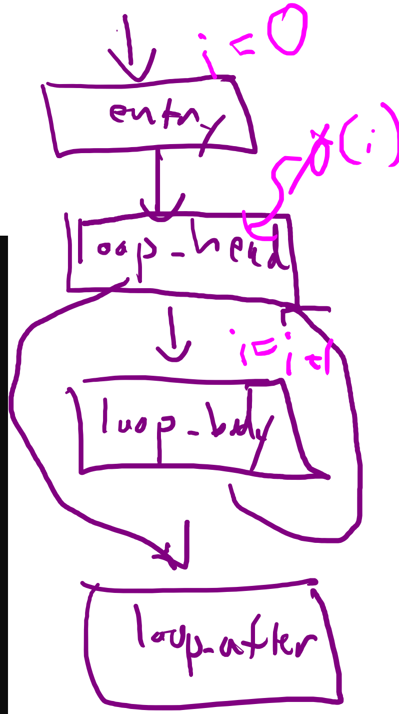
LLVM REGISTER OPERATION REVIEWS

Write your name and answer the following on a piece of paper

- *Write out the corresponding LLVM bitcode program for the following:*

```
int main(int argc){
  int i = 0;
  while (i < argc){
    i = i + 1;
  }
  return i;
}
```

```
define i32 @main(i32 %argc) #0 {
entry:
        %i_init = add i32 0, 0
        br label %loop_head
loop_head:
        %i_join = phi i32 [%i_init, %entry], [%i_loop, %loop_body]
        %done = icmp slt i32 %i_join, %argc
        br i1 %done, label %loop_body, label %loop_after
loop_body:
        %i_loop = add i32 %i_join, 1
        br label %loop_head
loop_after:
        ret i32 %i_join
}
```



ADMINISTRIVIA AND ANNOUNCEMENTS

We have to talk about cheating

LLVM BITCODE MEMORY

EECS 677: Software Security Evaluation

Drew Davidson

CLASS PROGRESS

WE’RE GEARING UP TO BUILD OUR OWN PROGRAM ANALYSES

- WE HAVE THE THEORY FOR DATAFLOW
- WORKING THROUGH A GOOD PROGRAM REPRESENTATION

LAST TIME: LLVM BITCODE & REGISTERS

REVIEW: LAST LECTURE

LOW-LEVEL LANGUAGE

- Targets an abstract machine
- Uses a system of (infinite) named registers to perform computation
- Registers must be in SSA format

LECTURE OUTLINE

- LLVM Memory
- Load/Store
- The dreaded GEP

LLVM MEMORY

LLVM BITCODE

ENCODES THE CONCEPTS OF LOCAL AND GLOBAL MEMORY

Local memory: within a function activation

Global memory: static in the data section

Notably absent: heap memory

With infinite registers,
Why have local memory?

Because we might take the address of a local

```
int main(){
  int a;
  int * p;
  p = &a;
}
```

p points to a

LLVM MEMORY: ALLOCATION

LLVM BITCODE

Allocating Global Memory

```
@glb1 = global i32 2, align 4
@cnst2 = constant i32 3, align 4
```

Allocating Local Memory

```
%reg = alloca i32, align 4
```

LLVM MEMORY: LOAD AND STORE

LLVM BITCODE

STORE

```
store <srcType> <srcOpd>, <dstType> <dstOpd>, align <align>
store    i32          1    ,   i32*      %var1ptr, align   4
```

LOAD

```
<dstOpd> = load <dstType>, <srcType> <srcOpd>, align <align>
  %reg   = load    i32,        i32*   %var1ptr, align 4
```

```
int main(){
  int val;
  val = 12;
  return val;
}
```

```
define i32 @main() #0 {
  %valptr = alloca i32, align 4
  store i32 12, i32* %valptr, align 4
  %res = load i32, i32* %valptr, align 4
  ret i32 %res
}
```

LLVM MEMORY: GLOBAL MEMORY EXAMPLE

LLVM BITCODE

```c
int a = 2;
int main(){
    return a;
}
```

```llvm
@a = global i32 2, align 4
define i32 @main() #0 {
  %reg = load i32, i32* @a, align 4
  ret i32 %reg
}
```

LLVM MEMORY: LOOK, NO SSA!

LLVM BITCODE

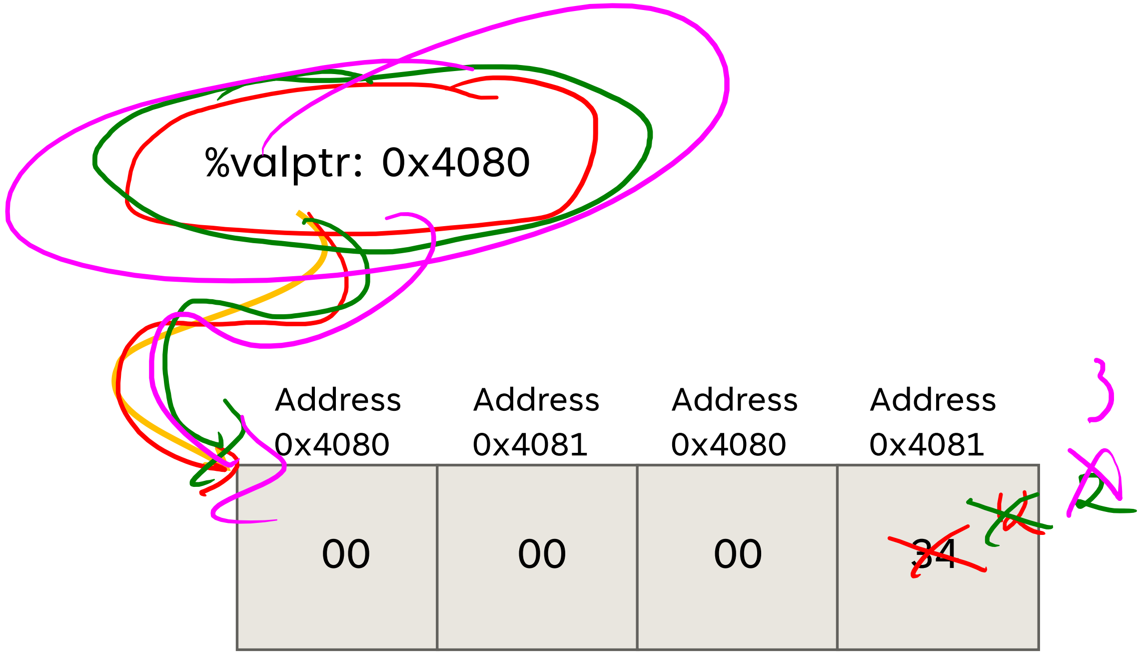
```
define i32 @main() #0 {
  %valptr = alloca i32, align 4
  store i32 12, i32* %valptr, align 4
  store i32 2, i32* %valptr, align 4
  store i32 3, i32* %valptr, align 4
  %res = load i32, i32* %valptr, align 4
  ret i32 %res
}
```

The VALUE OF the register doesn't change
The VALUE AT the register is what changes!

LLVM MEMORY: LOOK, NO SSA!

LLVM BITCODE

```
define i32 @main() #0 {
  %valptr = alloca i32, align 4
  store i32 12, i32* %valptr, align 4
  store i32 2, i32* %valptr, align 4
  store i32 3, i32* %valptr, align 4
  %res = load i32, i32* %valptr, align 4
  ret i32 %res
}
```



LLVM MEMORY: AGGREGATE TYPES

LLVM BITCODE

RECALL THAT BITCODE IS A TYPED LANGUAGE

Declare an aggregate type (think struct)

%Point = type { i32, i32 }

Allocate an aggregate type

%ptr = alloca %Point, align 4

Allocate an array

%arrayptr = alloca [8 x i32], align 16

%ArrSize8 = type [8 x 32]

%Struct2 = type { i32, %ArrSize8 }

LLVM Memory: Accessing Aggregate Memory

LLVM Bitcode

At this point, we need to discuss how to read an array index or Field

There is a powerful, but somewhat complicated instruction to do it, called getelementptr (GEP)

GEP never actually reads memory, it just computes what the offset from a base location would be

LECTURE OUTLINE

- LLVM Memory
- Load/Store
- The dreaded GEP

GETELEMENTPTR

LLVM BITCODE

Here is the Basic format of a GEP

```
<result> = getelementptr <ty>, ptr <ptrval>{, [inrange] <ty> <idx>}*
```

Here is a snippet of the documentation of the syntax:

The first argument is always a type used as the basis for the calculations. The second argument is always a pointer or a vector of pointers, and is the base address to start from. The remaining arguments are indices that indicate which of the elements of the aggregate object are indexed. The interpretation of each index is dependent on the type being indexed into. The first index always indexes the pointer value given as the second argument, the second index indexes a value of the type pointed to (not necessarily the value directly pointed to, since the first index can be non-zero), etc. The first type indexed into must be a pointer value, subsequent types can be arrays, vectors, and structs. Note that subsequent types being indexed into can never be pointers, since that would require loading the pointer before continuing calculation.

GETELEMENTPTR

LLVM BITCODE

LET ME (MAYBE?) SIMPLIFY THIS A BIT WITH A CONSTRAINED VERSION OF GEP

<result> = getelementptr <ty_{res}>, <ty_{srcobj}> <srcobj>, <ptrtype> <siblingidx>, [<ptrtype> <fieldidx>]+

HERE IS MY EXPLANATION OF THIS VERSION OF GEP:

Get a *pointer* of type <ty_{res}> by...
- starting from the base address srcobj
- jumping over siblingidx siblings
- jumping over fieldidx children

GETELEMENTPTR: PICTORIALLY

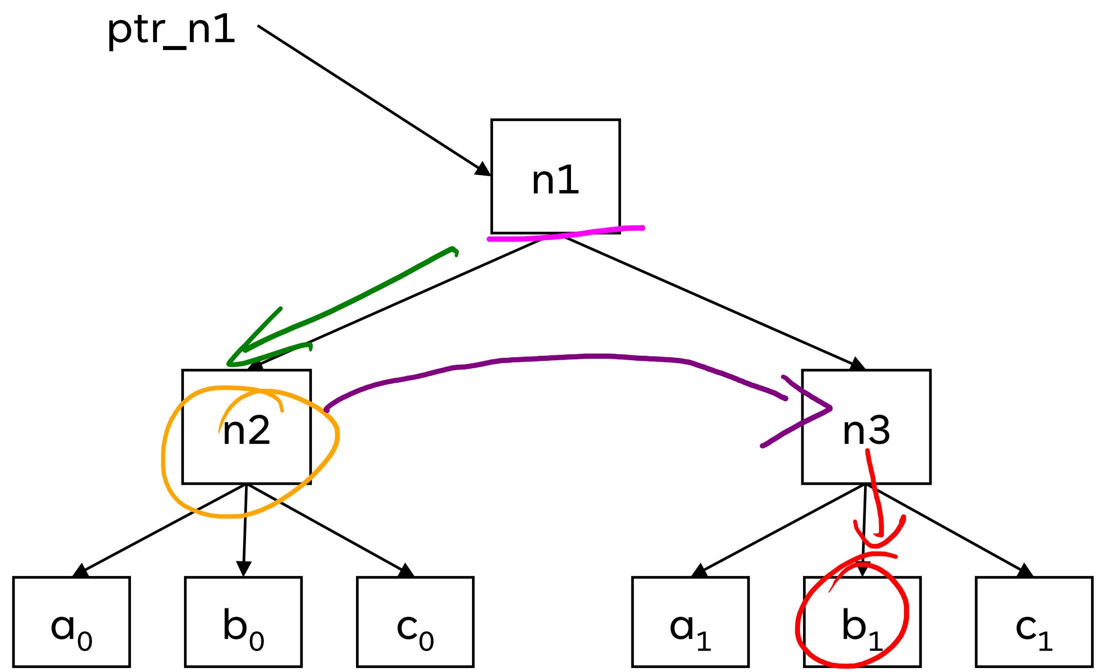
LLVM BITCODE

Can be helpful to walk through memory as a tree

```
%t1 = type { A, B, C }
%t2 = type [ 2 x %t1 ]
@ptr_n1 = global %t2 [ { a0, b0, c0 }, { a1, b1, c1} ]

ptr_n2 = getelementptr %t2* ptr_n1, i64 0, i64 0

ptr_b1 = getelementptr %t2* ptr_n2, i64 1, i64 1
```



WRAP-UP

NEXT TIME

A COUPLE MORE BITCODE FEATURES

DESCRIBE THE FIRST PROJECT