

CS 536

Runtime Access to Variables

Quick Survey: Printing out Slides

- Printed slides can be a good reference
- Present limitations in Powerpoint



No more Anonymous Piazza

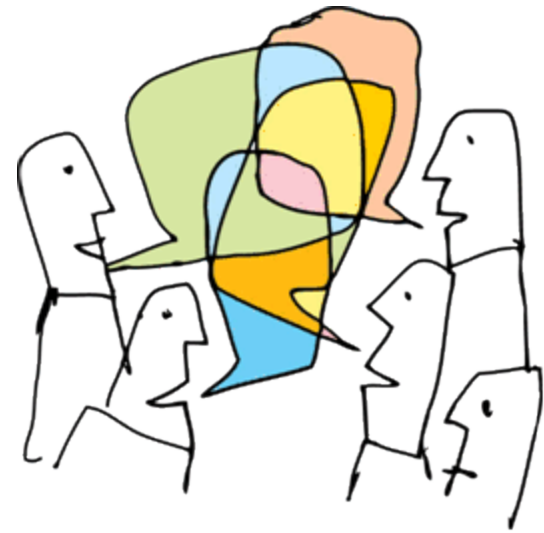
- Some abuse directed towards a student on the midterm survey
- Apologies for not catching this sooner



THIS IS WHY WE CAN'T
HAVE NICE THINGS

About that Discussion Section...

- How many people would attend?



Roadmap

- Last Time
 - Discussed runtime environments
 - Described some conventions for assembly
 - Functions via a stack
 - Dynamic memory via a heap (not used in C-Flat)
- This Time
 - Representing variables

Outline

- Scope levels
- A little bit of MIPS for accessing variables
- How we set up locals

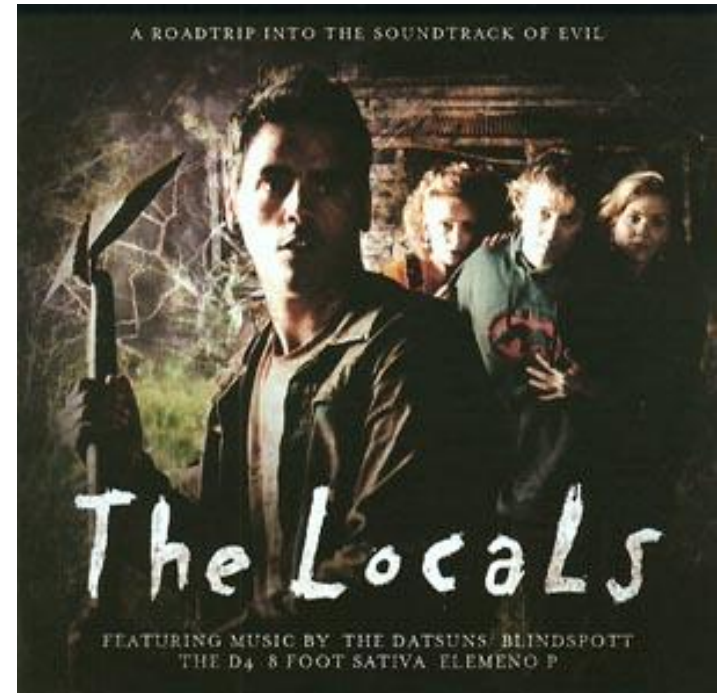
Scope

- We mostly worry about 3 flavors
 - Local
 - Declared and used in the same function
 - Further divided into “block” scope in C-Flat: declared in the same block
 - Global
 - Declared at the outermost level of the program
 - Non-local
 - For static scope: variables declared in an outer nested sub-program
 - For dynamic scope: variables declared in the calling context




Local variables

- Includes params, declarations in all block scopes
- Stored in the AR of the current function



Local variables: Examples

- What are the local variables here?

```
int fun(int a, int b) {  
    int c;  
    c = 1;  
    if (a == 0) {  
        int d;   
        d = 4;  
    }  
}
```

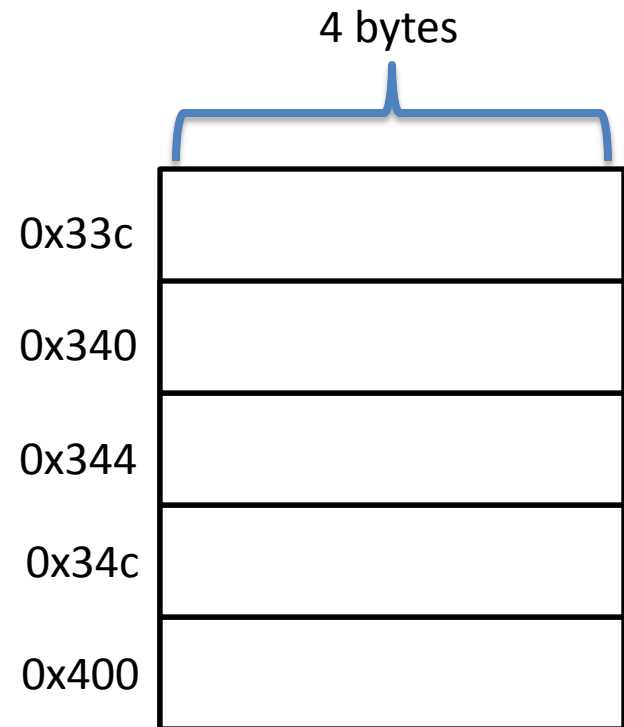
How do we access the Stack?

- Need a little MIPS knowledge
 - Full tutorial on Thursday
 - General anatomy of a MIPS instruction

opcode Operand1 Operand2

How do we access the Stack?

- Use “load” and “store” instructions
 - Recall that every memory cell has an address
 - Calculate that memory address, then move data from/to that address



MIPS mini-tutorial

- Need a little MIPS knowledge
 - Full tutorial on Thursday
 - General anatomy of a MIPS instruction

Basic memory operations

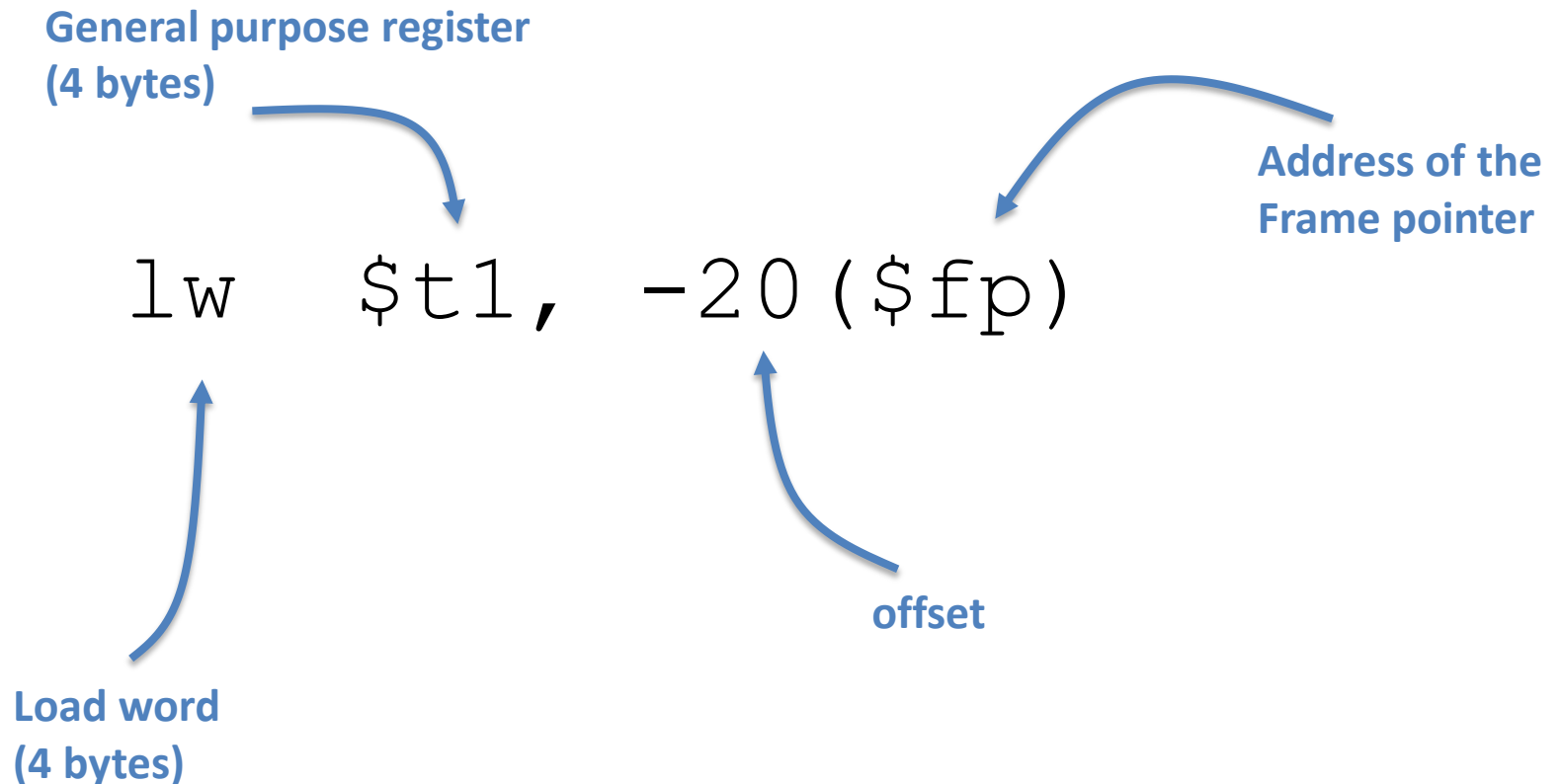
- Note to x86 cognoscenti: no fixed “destination” operation and

```
lw register memoryAddress
```

```
sw register memoryAddress
```

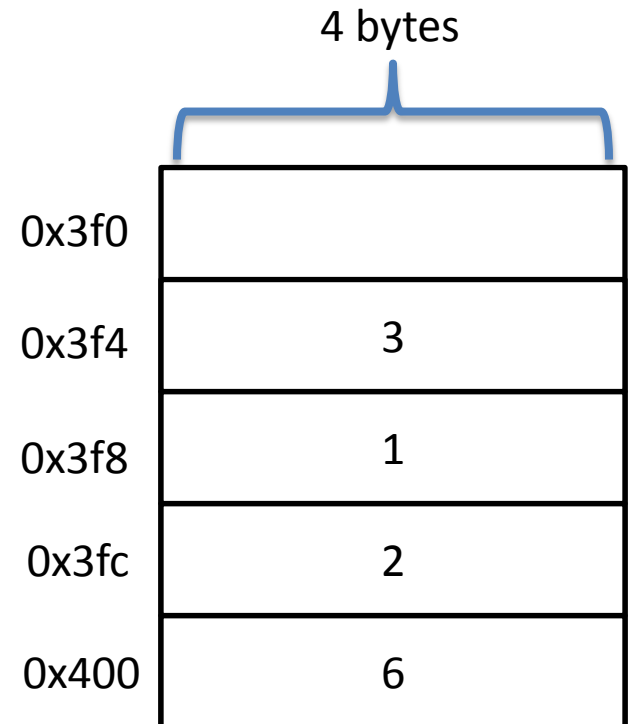
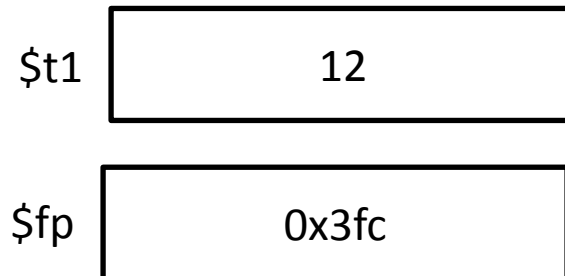
Load Word Example

opcode register memoryAddress



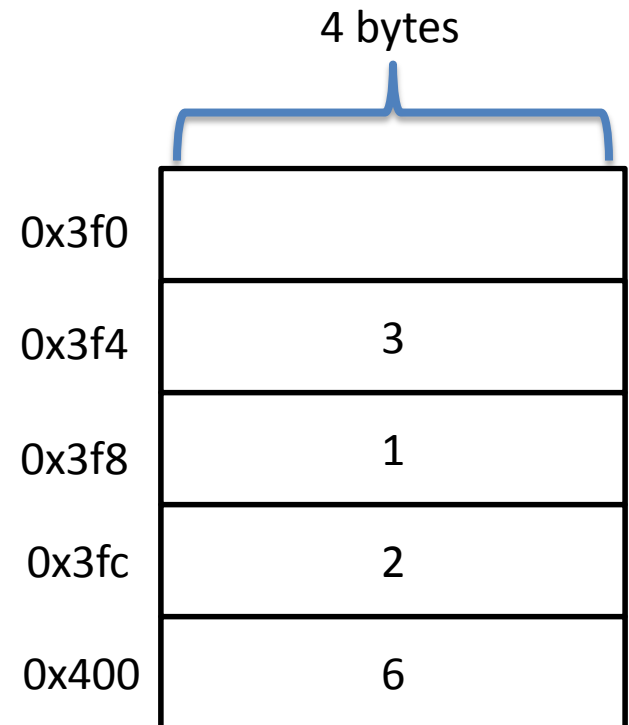
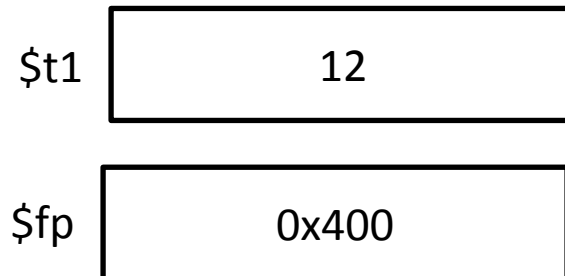
Load Word in Action

```
lw    $t1, -8($fp)
```



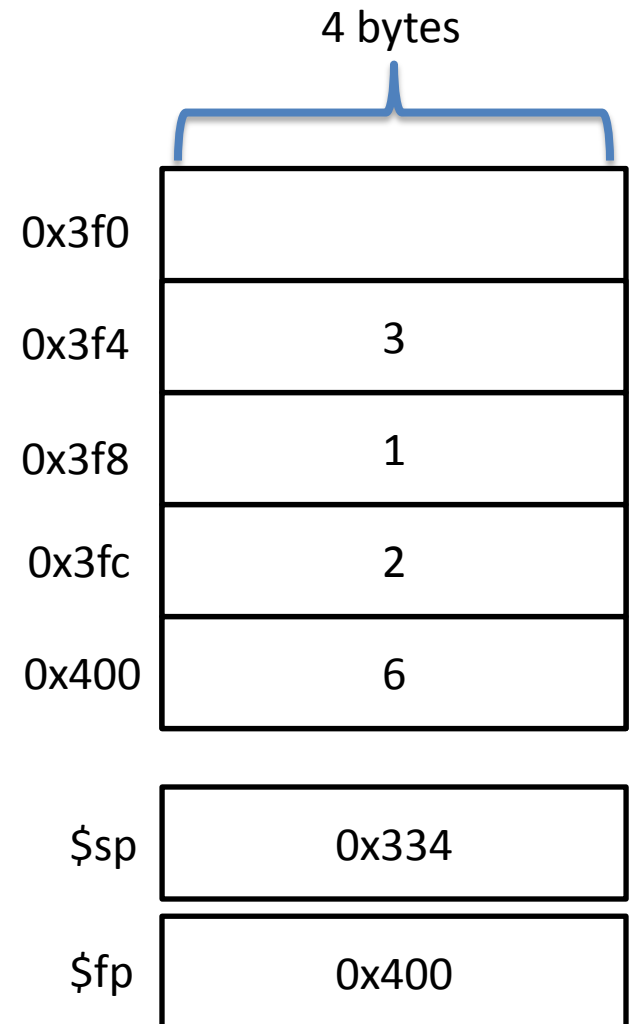
Store Word in Action

```
sw    $t1, 0($fp)
```



Relative Access for Locals

- Why do we access locals from \$fp?
 - That's where the activation record starts
- What if we used \$sp instead?



Simple Memory-Allocation Scheme

- Reserve a slot for each variable in the function

```
int test (int x, int y) {  
    int a, b;  
    if (x) {  
        int s;  
    } else {  
        int t, u, v;  
        u = b + y;  
    }  
}
```

\$sp	0x3d0
\$fp	0x400

0x3d4	
0x3dc	(v)
0x3e0	(u)
0x3e4	(t)
0x3e8	(s)
0x3ec	(b)
0x3f0	(a)
0x3f4	(control link)
0x3f8	(return addr)
0x3fc	(y)
0x400	(x)

Simple Memory-Allocation Algorithm

For each function

Set offset = 0

for each parameter

 add name to symbol table

 offset -= size of parameter

offset -= size of return address

offset -= size of control link

offset -= size of callee saved registers

for each local

 add name to symbol table

 offset -= size of variable

Simple Memory-Allocation Implementation

- Add an offset field to each symbol table entry
- During name analysis, add the offset along with the name (Wait until Project 6 to do this)
- Walk the AST performing decrements at each declaration node

Algorithm Example

```
int test (int x, int y) {  
    int a, b;  
    if (x) {  
        int s;  
    } else {  
        int t, u, v;  
        u = b + y;  
    }  
}
```

-36	(v)
-32	(u)
-28	(t)
-24	(s)
-20	(b)
-16	(a)
-12	(control link)
-8	(return addr)
-4	(y)
0	(x)

What about Global Variables?

- In a sense, easier to handle than locals
 - Space allocated directly at compile time (instead of indirectly via \$fp and \$sp registers)
 - Never needs to be deallocated
- MIPS actually keeps data in a completely different area of memory
 - Special assembler directives define memory regions

Handling Global Variables

- In a sense, globals easier to handle than locals
 - Space allocated directly at compile time (instead of indirectly via **\$fp** and **\$sp** registers)
 - Never needs to be deallocated
- Place in static data area
 - In MIPS, handling with a special storage directive
 - Variables referred to by name, not by address

Memory Region Example

```
.data
_x: .word 10
_y: .byte 1
_z: .asciiz "I am a string"
.text
lw $t0, _x    #Load from x into $t0
sw $t0, _x    #Store from $t0 into x
```


Accessing non-local variables

- Static scope
 - Variable declared in one procedure and accessed in a nested one
- Dynamic scope
 - Any variable use not locally declared

Static non-local scope example

- Each function has it's own AR
 - Inner function accesses the outer AR

```
function main() {  
    a = 0;  
    function subprog() {  
        a = a + 1;  
    }  
}
```

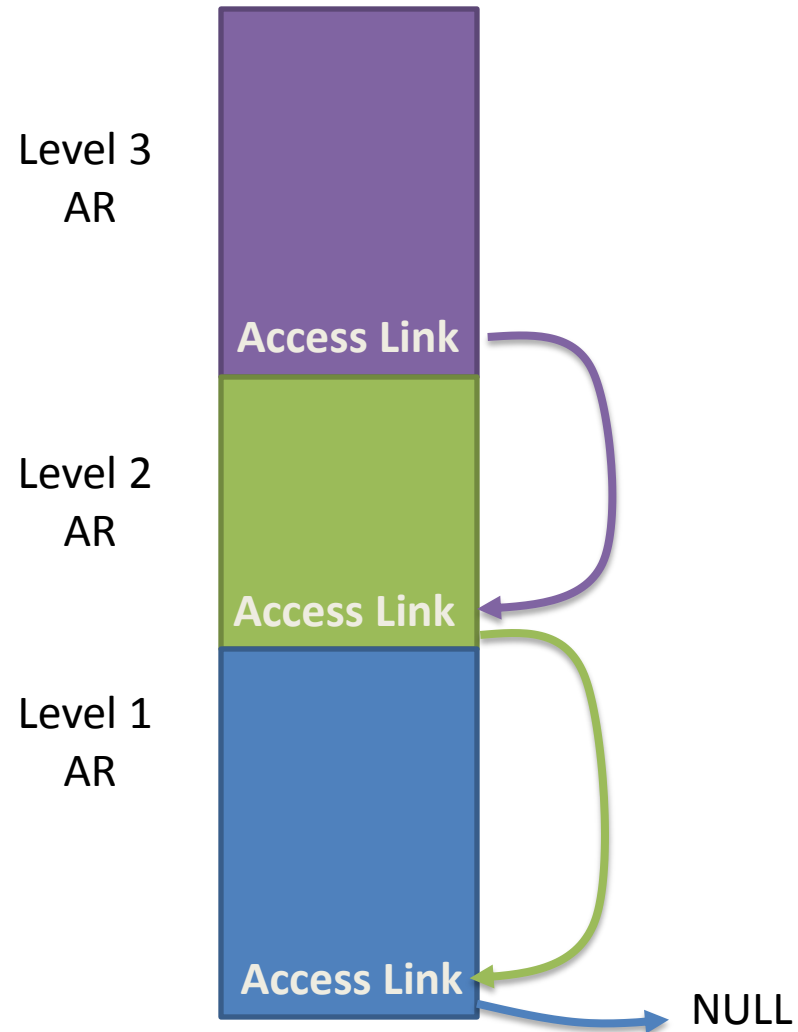
Static non-local scope memory access

```
void procA(){ // level 1
    int x, y;       $x_1$ 
    void procB(){ // level 2

        void procC(){ //level 3
            int z;
            void procD(){
                int x;
                x = z + y;
                procB();
            }
            x = 4;
            z = 2;
            procB();
            procD();
        }
        x = 3;
        y = 5;
    }
}
```

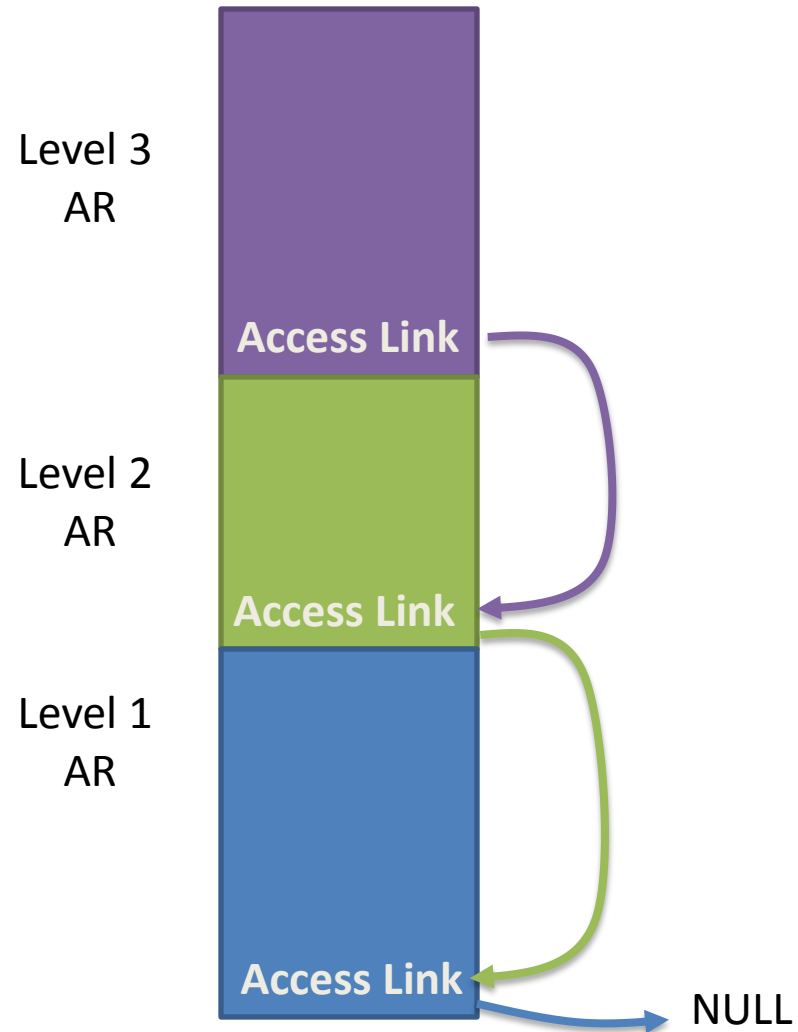
Access Links

- Add an additional field to the AR
 - Points to the locals area of the outer function
 - Sometimes called the static link (since it refers to the static nesting)



How Access Links Work

- We know how many *levels* to traverse statically
 - Example: In nesting level 3 and the variable is in nesting level 1: go back access links $(3 - 1)$ 2 levels



Setting up access links

- Using 1 access link

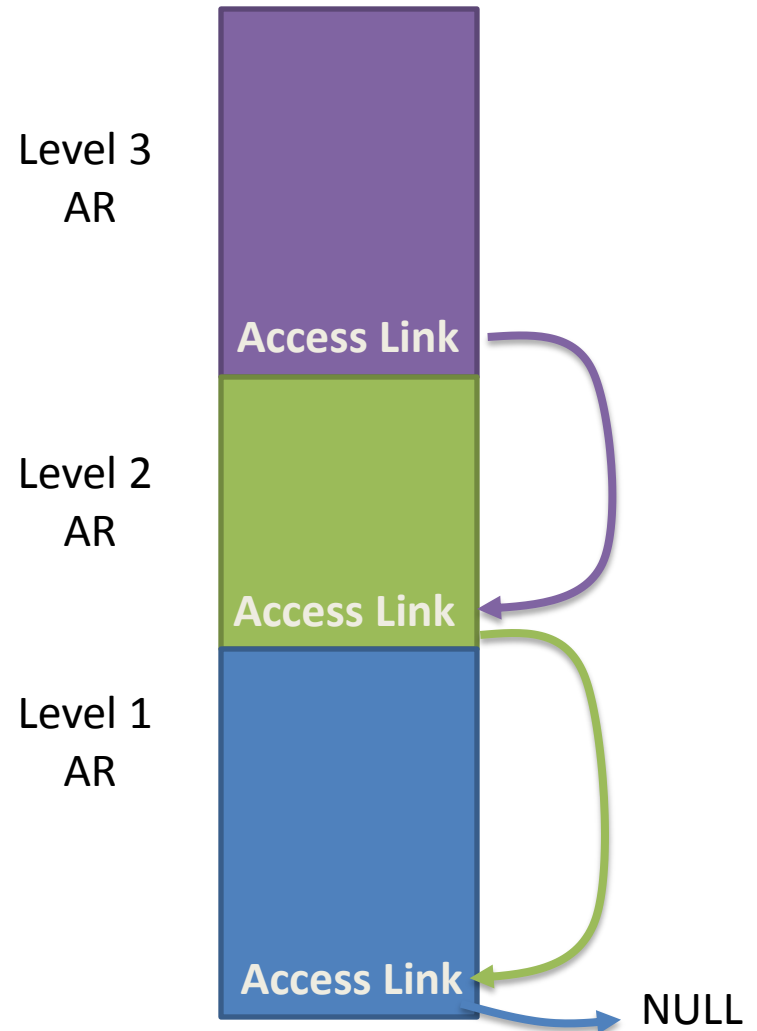
```
lw $t0, -4($fp)
lw $t0, -12($t0)
```

Where

\$fp -4 is the location of the access link
the variable in the outer scope
at offset 12 from outer AR

- Using 2 access links

```
lw $t0, -4($fp)
lw $t0, ($t0)
lw $t0, -12($t0)
```



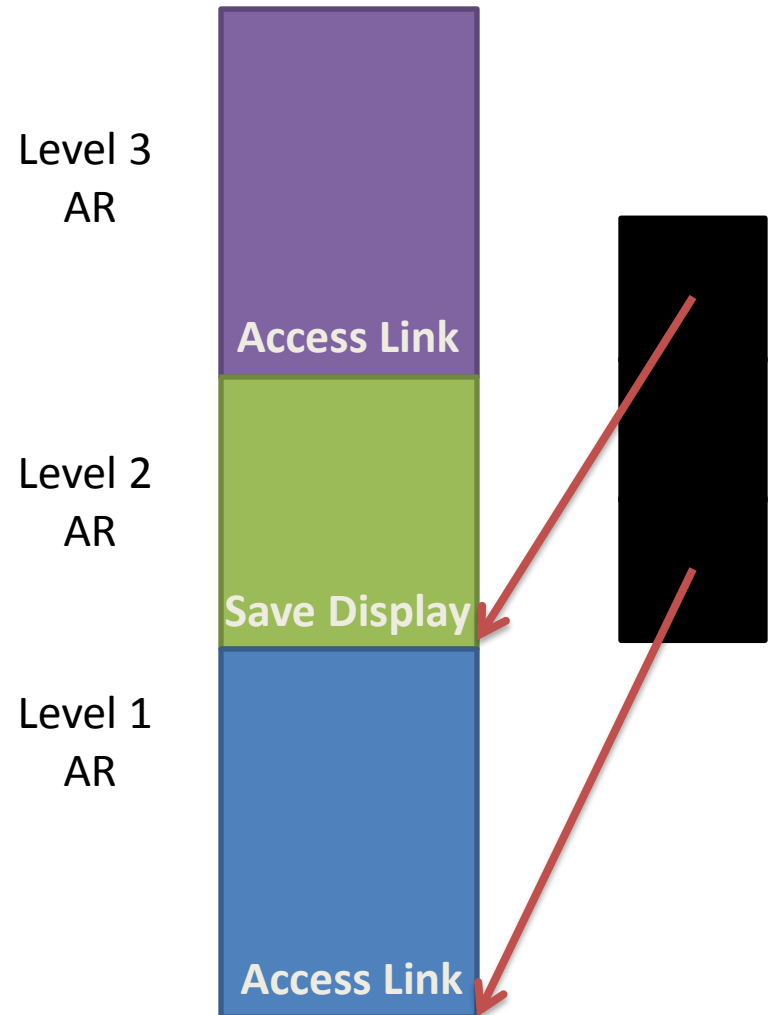
Thinking about access links

- We know the variable we want to access statically
 - Why don't we just index into the parent's AR using a large positive offset from \$fp?

```
lw $t0 38($fp)
```

Displays

- High-level idea:
 - Keep the transitive effects of multiple access link traversals
 - Uses a side-table of this info
- Access Link tradeoffs
 - Faster to call far up the heirarchy
 - Takes extra space



Questions about Static Scope?

Dynamic non-local scope example

```
function main() {  
    a = 0;  
    fun();  
}
```

```
function fun() {  
    a = a + 1;  
}
```

Dynamic Scope Storage

- Key point
 - We don't know *which* non-local variable we are referring to
- Two ways to set up dynamic access
 1. Deep Access – somewhat similar to Access links
 2. Shallow Access – somewhat similar to displays

Deep Access

- If the variable isn't local
 - Follow the control link to the caller's AR
 - Check to see if it defines the variable
 - If not, follow the next control link down the stack
- Note that we somehow need to know if a variable is defined *by name* in an AR
 - Usually means we'll have to associate a name with a stack slot

Shallow Access

- Keep a table with an entry for each variable declaration
 - Compile a direct reference to that entry
 - At a function call
 - Save all locals in the *caller's* AR
 - Restore locals when the *callee* is finished