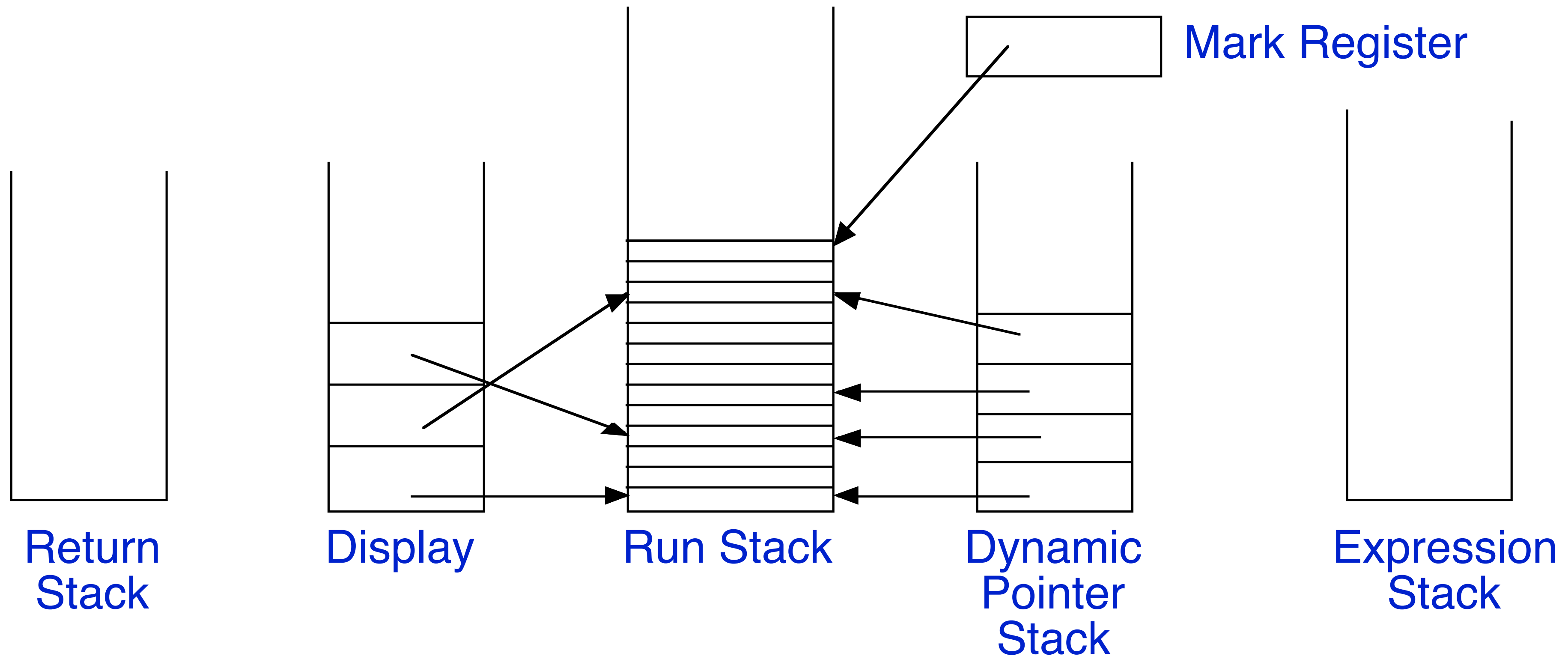
Last Time

- Two tasks left - storage allocation and code generation
- In order to allocate storage and generate code for a target machine, we need to choose representations of the abstract machine structures - this depends on the target machine
- Choice of representation is the single biggest factor in determining the quality of code we can generate
- There are several classes of target machines, for which we need to make different representation decisions - stack machines, register machines, RISC machines, fixed vs windowed registers

Recall ...

- Run Time Model – 5 stacks
  - Expression Stack – expression evaluation
  - Run Stack – storage allocation
  - Display – scope management
  - Dynamic Pointer Stack – restoring Run Stack and Display
  - Return Stack – remember program counter for call/return



Representing the RunTime Model

- Representations of the *ES*, *RS* and *Display* are the most critical because they are the most used (in every expression, every variable reference)

- This time we look at representing these on different classes of machines

- Keep in mind we are facing tradeoffs - often the easiest or most obvious choice yields the worst generated code!

Implementing the Expression Stack

- Our task is to map the structures of the abstract machine to the registers, stacks, memory, instructions and operands of the target computer

- Recall that the Expression Stack is the stack used to evaluate the values of all expressions, and to do most computation - so we should choose to represent it using the most efficient resources of the target computer

- If the target computer has no registers and a fast built-in hardware stack (e.g., Burroughs B5500) then this is easy - we just represent the ES using the hardware stack, and we are done

Implementing the Expression Stack

- If the target computer has many general purpose registers (e.g., VAX, IBM/360, all RISC computers), then we can implement the ES using a subset of registers that we manage as if it were a stack

- Example: IBM/360 (Z-series) mainframe:

```
pushaddress a          LA     R1,a
push b                 L      R2,b
push c                 L      R3,c
add                    AR     R3,R2
assign                 ST     R2,0(R1)
```

- In practice the ES rarely goes deeper than about 3 or 4 deep, so we can use just 3 or 4 registers - but since we need to handle any expression (otherwise we're not a very good compiler), we must be prepared to either arrange that we never use more than the 3 or 4 (this is provably possible) or we need to have a backup strategy

- The backup strategy typically reserves some space on a memory stack or in global memory to represent elements of the ES that overflow

Implementing the Expression Stack

- If the target computer has few registers, but also has a memory stack (e.g., PDP-11, Intel x86), then there are often not enough registers to implement the entire ES

- In this case we can allocate only one or two registers for the ES, and arrange that the registers represent the top two elements of the ES while the lower elements are pushed on the memory stack

- On Intel x86 computers registers are particularly scarce, so compilers often use an accumulator model, where only the top element of the ES is represented in a register - this model works like a simple calculator, where the current display value is operated on by each operation

- On these machines if speed is less important and space is tight, then the hardware stack is used exclusively (no registers) - this is only possible on architectures that permit direct memory to memory operations (e.g., PDP-11, VAX, NS32000)

Implementing the Run Stack - Static Variables

- Recall that in our abstract machine model, the Run Stack represents the storage for all variables in the program

- Static variables are global variables that persist the entire time the program is running - these are the variables at LL0

- Most computers have the idea of a global data segment and all have addressing modes that allow access to static addresses in memory

- Some have addressing modes to access memory at absolute addresses (e.g., VAX, PDP-11, MC68000, PowerPC, x86) - on these computers we can simply represent LL0 variables at permanent absolute addresses

- Example: PDP-11

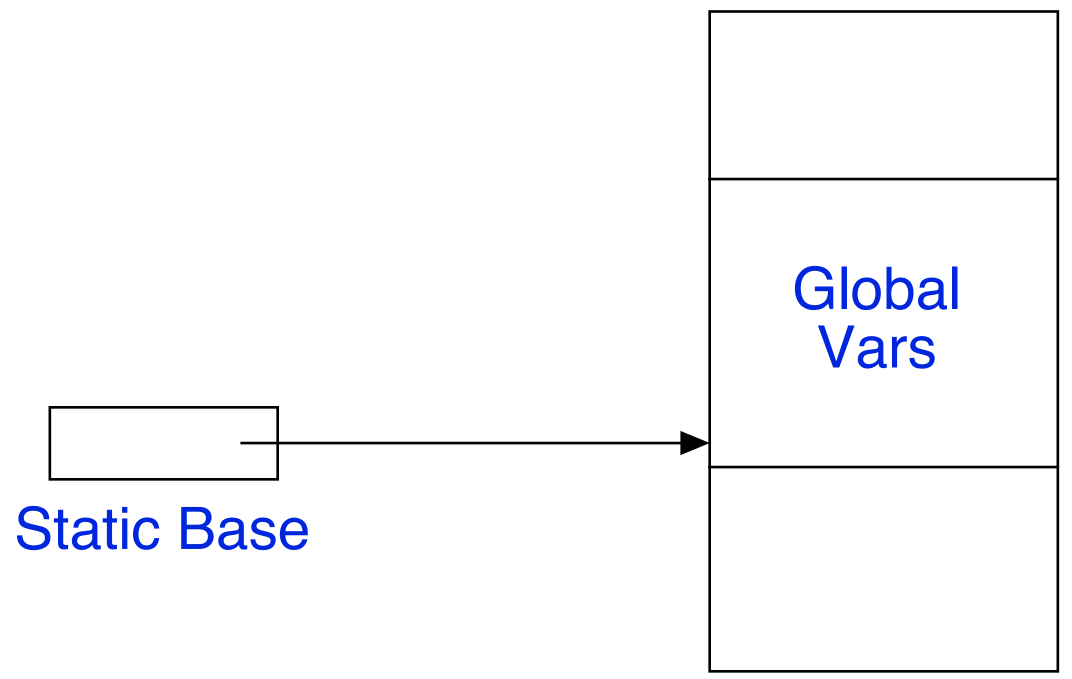
```
         .data                      .text
    a:   .word   0                  mov     a,r1
```

Implementing the Run Stack - Static Variables

- Other computers use a dedicated register called the static base to point to the start of the global variable space in memory, and all access to global variables is indirect through the static base

- Examples of machines using this technique are the IBM/360 (which may have multiple static bases) and the NS32000

- One way to envision this is that the static base register simply represents Display[0]



Implementing the RS - Automatic Variables

- Automatic variables require that we actually implement a memory stack to represent the Run Stack - if the architecture does not provide a memory stack, then we build one using code

- Many architectures actually have a memory stack concept built in (e.g., PDP-11, x86, VAX, PowerPC, etc.) - in these architectures, we (obviously) use the stack for automatic variables

- For machines without a stack (e.g., IBM/360), we allocate a section of memory to serve as the stack, and dedicate a general purpose register to act as the stack pointer - push and pop from the stack is implemented by generating instructions to add or subtract from the stack pointer register

- The direction that the stack operates is not the same on all architectures - on most architectures the stack is at the top of memory and grows down, but on others, the reverse is true

Implementing the Display

- If our implementation of Run Stack storage is to be effective, we must also choose an implementation of the Display (so that (LL,ON) addressing can be implemented)

- Because we implement global variables (those at LL0) in static memory, we don't need to represent Display[0]

- For some languages (such as Fortan and C) we need only support two lexical levels - LL0 in static memory, and LL1, the current procedure's local variables, on the stack - in this case there is no need to represent the Display at all, if we use the stack pointer itself as a (backwards) base (this trick is used by most C compilers)

- If the programming language does not support recursion, as is the case for PT Pascal, then there can be at most one instance of the local variables of a procedure or module - this means that <u>all</u> variables can be allocated statically as globals, and there is no need for a memory stack or Display at all!

Implementing the Display

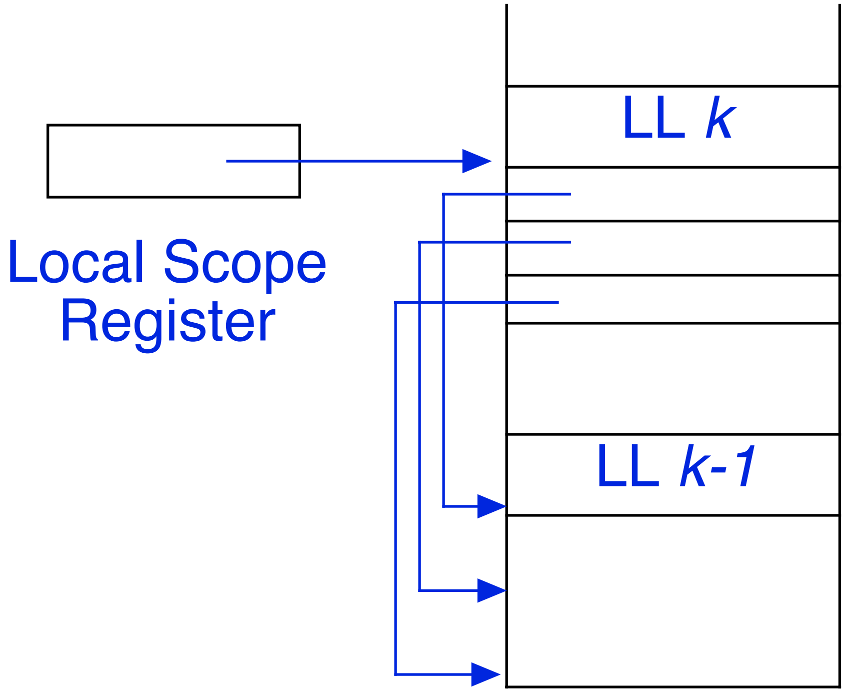
- For the more general case (full Pascal, PL/I, ADA, etc.) the Display must somehow be mapped to the target hardware
- Two general techniques are used: Display registers and stack based Displays
- At compile time, we know the maximum lexical level used in the program (the deepest that code is nested) - if our machine has plenty of registers, we can dedicate some of them (Max LL - 1) to represent the elements of the Display - we call these the Display registers - this is currently the most common technique
- On machines with a limited number of registers we can either impose an implementation limit on the depth of nesting we will permit (not a very good solution) or we can represent the Display as an array in memory and load Display entries into a register as we need them
- To save the previous value of a Display pointer on entry to a procedure, we push the previous value on the stack as an extra local variable and restore it when we return

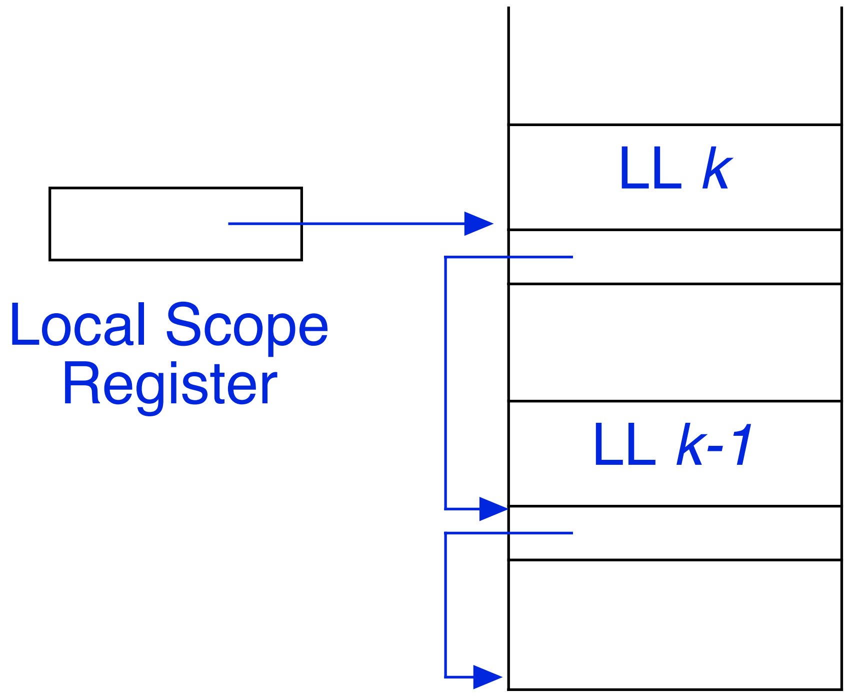
Implementing the Display

There are a variety of ways in which the Display can be represented as part of the memory stack itself

One way is to represent the entire Display at the bottom of the current local stack frame in memory - one register locates the stack frame, and all other Display entries can be fetched from there

Another way is to chain the Display entries together, so that each scope on the stack has a pointer to the scope one LL down





Summary

- In order to allocate storage and generate code for a target machine, we must first choose representations of the abstract machine structures

- The most critical decisions are the representations of the *ES*, the *RS* and the *Display* - these choices will largely determine the quality of code we can generate

- On most modern machines, both the *ES* and the *Display* are represented using registers, while the *RS* is represented using the machine's memory stack

- If the machine has no memory stack, we generate code to fake one using general purpose register as a pointer into a section of memory allocated to be our stack space

- If the machine has only a couple of registers, we use only one to locate the local scope and store the other Display entries in memory

Next Time

- **Next Week**
  - Quiz #3 - runtime model of scopes and storage, arrays and records, call/return, semantic analysis, the symbol and type stacks, semantic mechanisms, symbol tables, scope control

    (Chapters 12-16, Lectures 14-22, inclusive)

- **Then**
  - Storage Allocation & Code Generation