Code Generation Examples

Thanks to Ben Chelf and Jerry Cain for this handout.

Code Generation Quick Reference

The easiest way to deal with code generation is to deal with each line independently. Don't worry about optimizing code from line to line. Generate code for each line in the following way:

1) **Load all memory items into registers**
2) **Perform mathematical operations on values in registers**
3) **Store results from registers out to memory**

This will make your code easier for us to read and you to generate.

- **To generate code for dereferencing any kind of pointer, you load the memory at that pointer into a register.**

Ex: Generate code for `*intPointer`

```
R1 = Mem[SP]      ; Assume stack pointer is pointing to intPointer.
R2 = Mem[R1]      ; R2 now has the value intPointer was pointing to
```

- **To access structure or array members through a pointer, use the pointer as a base address for the array or structure. Access the array or structure members by adding an offset to the base address and then loading the memory at that offset.**

Ex: assume we have declared `char* buffer`.
Generate code for `buffer[3];`

```
R1 = Mem[SP]      ; Assume stack pointer pointing to buffer
R2 =.1 Mem[R1+3]  ; R2 now has the value at buffer[3]
```

Ex: assume we have declared the `struct`

```
struct foobar {
   int foo;
   int bar;
}
```

and a pointer to `foobar` is on the stack
Generate code for `foobar->bar;`

```
R1 = Mem[SP]      ; Assume stack pointer pointing to foobar pointer
R2 = Mem[R1+4]    ; ints are 4 bytes, so foobar->bar will be offset 4 bytes from the
```

base

- **To access the address of a local or parameter, just load the stack pointer offset into a register.**

  Ex: assume int `sum` is on the stack

  Generate code for `&sum;`

  ```
  R1 = SP          ; Assume stack pointer pointing to sum. That's it!
  ```

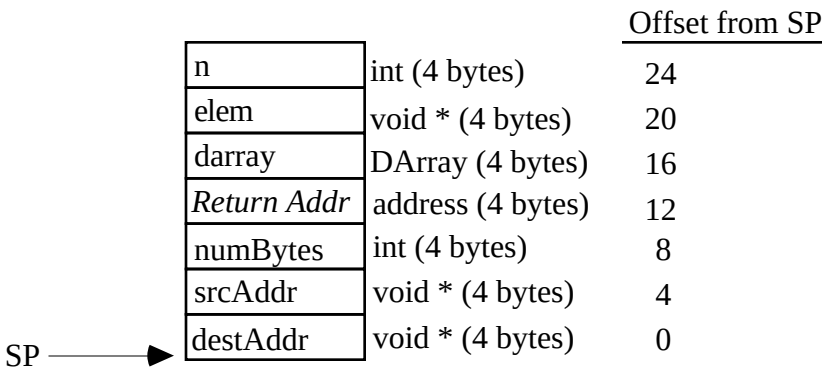
Code Generation

For the following function, (a) draw the activation record giving the size and type of each field, and then (b) generate code for each line of code. Assume that registers are **not preserved** in between individual lines of code.

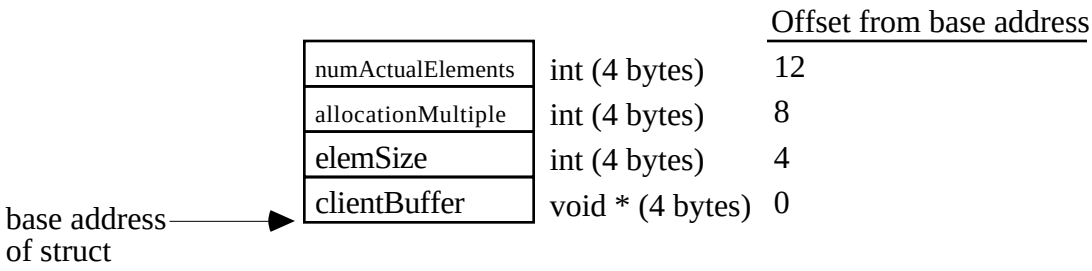
```
typedef struct {
   void  *clientBuffer;
   int   elemSize;
   int   allocationMultiple
   int   numActualElements;
} *DArray;

void BogusArrayInsertAt(DArray darray, void *elem, int n)
{
   void *destAddr, *srcAddr;
   int numBytes;

1  srcAddr = (void *)((char *) darray->clientBuffer + n*darray->elemSize);
2  destAddr = (void *)((DArray) srcAddr + darray->elemSize);
3  numBytes = (darray->allocationMultiple + n) * darray->elemSize;
4  darray->numActualElements *= 2;
}
```

Solution

The activation record for `BogusArrayInsertAt` looks like this:



It is also important to understand how the fields of the `struct` are laid out in memory with respect to its address. The memory layout is given as (next page please):



The generated code looks like:

```
SP = SP - 12      ; begin by making space for locals
```

Code for line 1: `srcAddr = (void *)((char *) darray->clientBuffer + n*darray->elemSize);`

```
R1 = Mem[SP+16]      ; load address of struct
R2 = Mem[R1]         ; load value of clientBuffer field (offset 0)
R3 = Mem[R1+4]       ; load value of elemSize field (offset 4)
R4 = Mem[SP+24]      ; load value of parameter n

R3 = R3 * R4         ; calculate the total offset in bytes
R2 = R2 + R3         ; add offset to array base

Mem[SP+4] = R2       ; store result in srcAddress field
```

Code for line 2: `destAddr = (void *)((DArray) srcAddr + darray->elemSize);`

```
R1 = Mem[SP+16]      ; load address of struct
R2 = Mem[R1+4]       ; load value of elemSize field (offset 0)
R3 = Mem[SP+4]       ; load value of srcAddress

R2 = R2 * 16         ; since the base address is cast to be DArray,
                     ; the offset stored in R2 should be interpreted
                     ; as the number of sizeof(*darray)-byte blocks...
R3 = R3 + R2         ; add offset to pointer

Mem[SP] = R3         ; store result in destAddress field
```

Code for line 3: `numBytes = (darray->allocationMultiple + n) * darray->elemSize;`

```
R1 = Mem[SP+16]      ; load address of struct
R2 = Mem[R1+4]       ; load value of elemSize field (offset 4)
R3 = Mem[R1+8]       ; load value of allocationMultiple field (off 8)
R4 = Mem[SP+24]      ; load value of parameter n

R3 = R3 + R4         ; calcluate the parenthesized sum
R2 = R2 * R3         ; calculate its product with elemsize

Mem[SP+8] = R2       ; store result in the numBytes variable
```

Code for line 4: `darray->numActualElements *= 2;`

```
R1 = Mem[SP+16]      ; load address of struct
R2 = Mem[R1+12]      ; load value of numActualElements field

R2 = R2 * 2          ; double the retrieved value

Mem[R1+12] = R2      ; and store back to numActualElements
```

And finally the cleanup and return code put in by the compiler:

```
SP = SP + 12      ; clean up space used for locals
RET               ; return to the caller
```

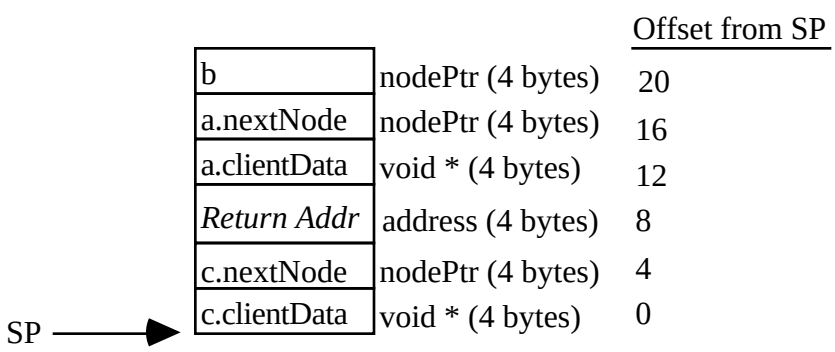
Advanced Code Generation

For the following function, (a) draw the activation record giving the size and type of each field, and then (b) generate code for each line of code. Once again assume that registers are **not preserved** in between individual lines of code.

```
typedef struct node {
   void *clientData;
   struct node *nextNode;
} node, *nodePtr;

void NoNonsensePantherNode(node a, nodePtr b)
{
   node c;

1  c.nextNode = b;
2  c.clientData = b;
   ……
3  *b = c;
   ……
4  b->nextNode = (nodePtr)((short *)&c + 2);
   ……
5  ((nodePtr)(b->clientData))->nextNode = NULL;
}
```

Solution:

The activation record for `NoNonsensePantherNode` looks like this:



The layout of a node struct in the heap should be evident given the layout of the node struct within the activation record.

First:

```
SP = SP – 8                    ;  Make space for local variables
```

Code for lines 1 and 2:

```
c.nextNode = b;
c.clientData = b;
```

```
R1 = Mem[SP+20]      ; load the value in b
Mem[SP+4] = R1       ; store b in c.nextNode
Mem[SP] = R1         ; store b in c.clientData
```

Code for line 3: `*b = c;`

```
R1 = Mem[SP+20]      ; load value of b (base address of struct)
R2 = Mem[SP]         ; load c.clientData field in register
R3 = Mem[SP+4]       ; load c.nextNode field in register

Mem[R1] = R2         ; store c.clientData in b->clientData
Mem[R1+4] = R3       ; store c.nextNode in b->nextNode
```

Note that our instruction set doesn't allow a load and a store within a single operation. So, although it may be **really, really** tempting, you may **not** include in the code things like:

```
Mem[R1] = Mem[SP]
Mem[R1+4] = Mem[SP+4]
```

Code for line 4: `b->nextNode = (nodePtr)((short *)&c + 2);`

```
R1 = SP           ; May seem strange, but SP is, after all,
                  ; the address of c.

R2 = 2 * 2        ; The 2 in the arithmetic expression should
                  ; be interpreted as 2 * sizeof(short) = 2 * 2.

R1 = R1 + R2      ; Add offset to pointer to finish RHS

R3 = Mem[SP+20]   ; Load b (the address of the struct we
                  ; are writing to).

Mem[R3+4] = R1    ; Write R1 to b->nextNode
```

Note that the above snippet of C code is functionally equivalent to

```
b->nextNode = (nodePtr)(&(c.nextNode));
```

Code for line 5: `((nodePtr)(b->clientData))->nextNode = NULL;`

```
R1 = Mem[SP+20];      ; Load address of the struct we are writing
                      ; to. Note that R1 is actually the address
                      ; of the clientData field

R1 = Mem[R1];         ; Load the address stored in the clientData
                      ; field; now that b->clientData is loaded,
                      ; pretend from here on that the value in
                      ; R1 is a nodePtr

Mem[R1+4] = 0         ; What? WHAT? well, the address stored in
                      ; R1 now needs to be interpreted as the
                      ; the base address of a node.  We know that
                      ; the address of the nextNode field is
                      ; then equal to R1 + 4, so we simply write
                      ; a zero to that address.

SP = SP + 8           ; Clean up locals
RET                   ; Finally, return to the caller
```