Computer Architecture: Take II

Handout written by Julie Zelenski and Nick Parlante

Example: Simple variables

A variable is a location in memory. When a variable is declared, the stated type determines how many bytes in memory are reserved for that variable. The compiler creates a symbol table to map between the symbolic name of a variable (e.g. "**i**") and the location in memory where it will be stored (e.g. address **6210**). We're not yet going to concern ourselves about exactly where in memory the compiler chooses to place variables. There is no run-time type information available about variables, so when reading or assigning to a location it is not clear whether you are working with an **int** or a **float** or a pointer, etc.

```
{
   int i;

   i = 6;
   i++;
}
```

In our code generation work, we will always assume **int**s are 4 bytes, so **i** has 4 bytes reserved for it, and let's say its address is currently stored in **R1**. Generated code:

```
M[R1] = 6      ; assign i the value 6

R2 = M[R1]     ; load i's value into a register
R2 = R2 + 1    ; do the addition
M[R1] = R2     ; store new value into i
```

The compilation translates one statement at a time. There is not a one-to-one correspondence between C statements and machine instructions; usually several low-level instructions are required to express the high-level statement. You can also see why compiled code tends to not run as fast as hand-coded CPU instructions ("assembly language"). A smart optimizing pass of the compiler could shorten the whole sequence to just **M[R1] = 7**.

When assigning or reading the value of variables that are less than full word size, the alternate form of load or store is used that indicates the number of bytes to move.

```
{
   char ch;

   ch = 'A';
}
```

ch has just one byte reserved since it is a **char**. Assume its address is stored in **R1**. Generated code:

```
M[R1] =.1 65   ; 65 is ASCII value of 'A'
```

Example: Type conversions

The types **char**, **short**, **int**, and long are all in the same family, and use the same binary polynomial representation. C allows you to freely assign between these types. When assigning from a smaller-sized type to a larger, there is no problem. All of the source bytes are copied and the remaining upper bytes in the destination are filled using what is called *sign extension*— the sign bit is extended across the extra bytes. This makes it so the upper bytes are set to all zero for positive numbers, all ones for negative numbers, which exactly replicates the original number but just in a larger number of bytes.

```
{
   char ch;
   int i;

   ch = 'A';
   i = ch;     // cast not needed, nor will it change result
}
```

Assume address of **i** is in **R1**, **ch** is at address **R1 + 4**. (i.e. just after **i**, later we'll learn that this is usually the way local variables are laid out). Generated code:

```
M[R1 + 4] =.1 65  ; assign ch ASCII value 'A'
R2 =.1 M[R1 + 4]  ; load ch into R2 (upper bytes of R2 are zeroed)
M[R1] = R2        ; assign value in R2 to i
```

Now **i** has the value 65 (which is the ASCII decimal representation for '**A**'), the three upper bytes are zeros. Assigning from a smaller to a larger data type never introduces any problems since any source value can be properly represented in the destination type.

It is not so easy in the reverse direction: an integer can hold a value that is out of range for a **short** or **char**. An assignment like this will only copy the lower bytes and ignores the upper bytes.

```
{
   char ch;
   int i;

   i = 1025;
   ch = i;  // a cast is not needed, nor will it change the result
}
```

Again, address of **i** is in **R1**, **ch** is at address **R1 + 4**. Generated code:

```
M[R1] = 1025         ; assign i the value 1025
R2 = M[R1]           ; load value of i into R2
M[R1 + 4] =.1 R2     ; copy lower byte of R2 to ch
```

The value **1025** is larger than the maximum value (**255**) that can be represented in a one byte **unsigned char**. When storing the value, the upper bytes are ignored, only the lower byte (which has value 1) is copied when assigning to the character. There's no mention of an overflow or a loss of information. (This is the kind of thing rocket explosions are made of...)

Example: More type conversions

As we discussed in lecture, the floating point types use a completely different representation scheme than the integer family of types. C allows you to assign between these types but the compiler inserts a special instruction that converts between the representations since the bit patterns are different.

```
{
   int i;
   float f;

   f = 3.14159
   i = f;
}
```

Address of **f** in **R1**, **i** at **R1 + 4**. Generated code:

```
M[R1] = 3.14159   ; assign f the value 3.14159

R2 = M[R1]        ; load value of f into R2
R3 = FtoI R2      ; convert value in R2 from float to int, store in R3
M[R1 + 4] = R3    ; copy value in R3 to i
```

In this case, **FtoI** is the instruction that takes a source register containing a float, converts to an integer and writes it to the destination register. There is also an **ItoF** instruction that converts in the other direction. Going from a float to an **int** truncates any fractional component. Even if the floating point number had no fractional part, the **FtoI** instruction will still read the IEEE floating point (mantissa + exponent) format and rewrite the value in the binary polynomial format of an integer. A floating point **3.0** is represented with a totally different bit pattern than the integer **3**.

Example: Control structures

CPU instructions are laid out in memory in sequential order and unless otherwise indicated, processing goes word by word from low addresses to higher addresses with no skipping or jumping. To implement a conditional code path for an **`if`**/**`then`** or **`loop`**, branch and jump instructions are used to change the sequence of instructions. Here is some code that conditionally executes a statement based on a comparison result:

```
{
   int i;

   if (i >= 0)
      i *= 2;
   i = 10;
}
```

Assume an instruction is encoded in one machine word (4 bytes), so successive instructions are 4 bytes away in memory. The branch and jump instructions refer to these addresses using a PC-relative scheme to route code through this passage. The address of **`i`** is in **`R1`**. Generated code:

```
R2 = M[R1]           ; load value of i into R2
BLT R2, 0, PC+12     ; if value < 0, skip to instr 12 bytes past current
R3 = R2 * 2          ; multiply i by 2
M[R1] = R3           ; store result back in i
M[R1] = 10           ; assign i constant value 10
```

When the value of **`i`** is not positive, the branch will jump over the instructions in the body of the "then" of the if statement, otherwise it will continue executing sequentially and meet up with the other path at a later instruction.

Loops are also constructed from branch and jump instructions. The bottom of the loop does an unconditional jump back up to the top of the loop. At the top, a conditional branch is tested and exits from the loop when the termination condition is detected:

```
{
   char ch;
   int i;

   for (i = 0; i < 10; i++)
      ch = 'a';      // silly, just to have something in loop body
   i = 25;
}
```

Address of **`i`** is in **`R1`**, **`ch`** is at address **`R1 + 4`**. Generated code:

```
M[R1] = 0            ; initialize i to value 0
R2 = M[R1]           ; load value of i into R2 (TOP of loop)
BGE R2, 10, PC+20    ; if value >= 10, get out of loop
M[R1 + 4]   =.1 97   ; store 'a' in ch
R2 = R2 + 1          ; add one to value of i
M[R1] = R2           ; store result back in i
```

```
JMP PC-20            ; jump back to top of loop (BOTTOM of loop)
M[R1] = 25           ; assign i constant value 25
```

A similar while loop, such as the one below, would compile to the same sequence of instructions.

```
{
   char ch;
   int i;

   i = 0;
   while (i < 10) {
      ch = 'a';
      i++;
   }
   i = 25;
}
```

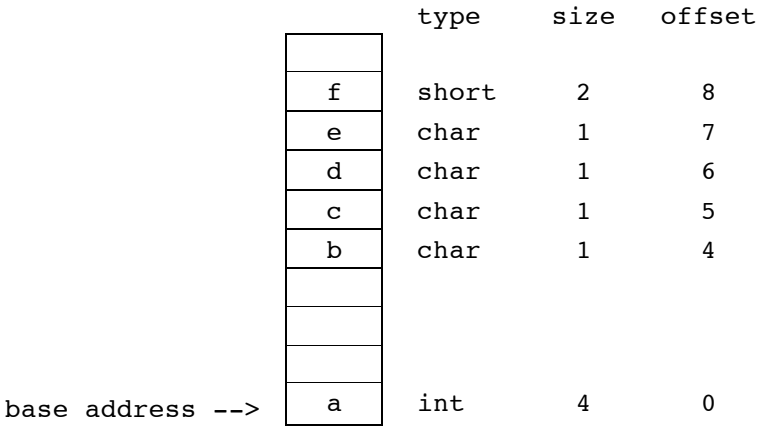
In fact, when you only have the generated code, all you can observe is the looping pattern with some termination condition— it is not at all apparent whether it was a `for` loop, while loop, or even something nasty constructed with **`goto`**s!

Example: Structures

The layout for a **`struct`** is determined by its compile-time type declaration. The compiler uses the **`typedef`** to calculate the size of the entire structure, as well as identify the type, sizes, and offsets of the component fields.

```
struct binky {
   int a;
   char b, c, d, e;
   short f;
};
```

Given the above definition, the structure would consist of 10 bytes laid out like this:



This code assigns a few of the fields in a **`struct binky`**:

```
{
   struct binky x;

   x.e = 'A';
   x.f = x.a;
}
```

Assumptions: there are **`10`** bytes allocated for the **`binky struct x`**, its base address is in **`R1`**.

Generated code:

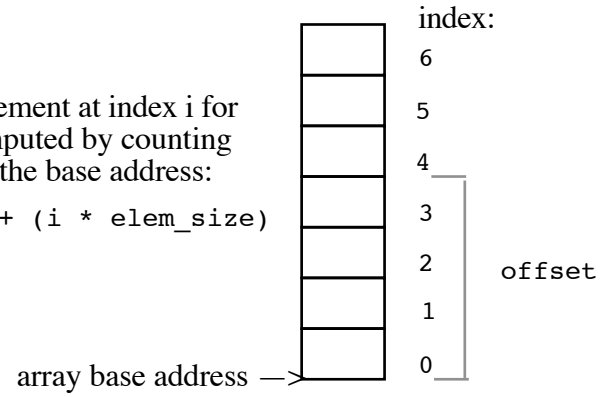
```
M[R1 + 7] =.1 65        ; R1 is the base addr of x, add 7 for .e offset
                        ; '=.1' means move one byte instead of 4
R2 = M[R1]              ; load value from .a which is at offset 0
M[R1 + 8] =.2 R2        ; store .f at offset 8, =.2 to move 2 bytes
                        ; will truncate value to the lower 2 bytes
```

Example: Arrays

Arrays are laid out contiguously in memory and every element is the same size. The compiler generates code to compute the address of any element by calculating that element's offset from the base of the whole array.

The location of the element at index i for 0-based arrays is computed by counting up "i" elements from the base address:

```
array_base_addr + (i * elem_size)
```



Consider a simple array of integers and an assignment to a particular entry (an index that is out of range, in fact. Does this produce any compile-time or run-time error?).

```
{
   int arr[20];

   arr[25] = 7;
}
```

Assumptions: **`arr`** has **`80`** bytes allocated (**`20 * sizeof(int)`**, the base address is in **`R1`**.

Generated code:

```
R2 = 25 * 4       ; compute offset (smart compiler can optimize)
R3 = R1 + R2      ; add to array base to get addr of 25th elem
M[R3] = 7         ; store into that location
```

Now consider this array of **50 binky struct**s and the C code to assign a field in one array element:

```
{
   int i;
   struct binky arr[50];

   a[i].f = i;
}
```

Assumptions: **arr** has 500 bytes allocated, its base address is in **R1**, **i** a 4-byte integer allocated at **R1 + 500** (i.e. just after the end of **arr**). Generated code:

```
R2 = M[R1 + 500]     ; load value of i
R3 = R2 * 10         ; multiply by element size to get offset
R3 = R1 + R3         ; add to array base addr to get addr of ith elem
M[R3 + 8] =.2 R2     ; don't forget the constant +8 offset for ".f"
```

Example: Pointers

A pointer is essentially an unsigned long which holds a memory address. Dereferencing a pointer means reading its value into a register and using it as the operand to a load instruction. Some C code:

```
{
   int *ptr;

   *ptr = 120;    // dangerous run-time behavior, but it compiles... =)
}
```

Assumptions: **ptr** allocated **4** bytes, its address is in **R1**.
Generated code:

```
R2 = M[R1]     ; load value of ptr
M[R2] = 120    ; dereference ptr and store 120
```

The **&** operator is a compile-time operation which looks up the given variable in the compiler's symbol table, determines where the variable is being stored, and then uses that address in the calculation. A little C snippet:

```
{
   int *s; i;

   s = &i;
   s++;
}
```

Assumptions: **i** allocated **4** bytes, its address is in **R1**, **s** at **R1** + **4**.
Generated code:

```
M[R1 + 4] = R1     ; store addr of i in s
R2 = M[R1 + 4]     ; load value of s into R2
R2 = R2 + 4        ; increment value by 4 (ptr to int!)
M[R1 + 4] = R2     ; store back into s
```

Note that the last three instructions look exactly the same as they would for adding 4 to some integer variable. When you look at the generated instruction stream, there is not nearly enough information to derive the original C source text. In fact, many different snippets of C code can compile down to the same sequence of machine instructions.

A pointer can also be used in pointer arithmetic expressions or with array bracket syntax. In both cases, it is important to realize that offsets from a pointer are always assumed to be in units of the base type. Adding **10** to a pointer or accessing the **10th** element does not translate to adding exactly **10** bytes to the base address; it instead adds **10 * sizeof(element)**.

```
{
   int *arr

   *(arr + 10) = 5;
   arr[10] = 5;        // these two lines compile to exact same sequence
}
```

Assumptions: **arr** address is in **R1**. Generated code (for one of the above statements, not both):

```
R2 = M[R1]         ; load base address (value of pointer arr)
R3 = 10 * 4        ; multiply offset by sizeof(int)
R4 = R2 + R3       ; add offset to base
M[R4] = 5          ; store at that location
```

Combining **struct**s and pointers, check out this code:

```
{
   struct binky **y, *x;

   x->f = 6;
   (**y).f = 7;
}
```

Assumptions: **x** address is in **R1**, **y** at **R1 + 4**. Generated code:

```
R2 = M[R1]         ; load x
M[R2 + 8] =.2 6    ; dereference x and offset by 8 to get .f
R2 = M[R1 + 4]     ; load y
R2 = M[R2]         ; dereference y once
M[R2 + 8] =.2 7    ; dereference again and offset by 8 to get .f
```

Example: Typecasts

A typecast is a compile-time entity that instructs the compiler to treat an expression differently than its declared type when generating code for that expression. For example, casting a pointer from one type to another could change the offset was multiplied for pointer arithmetic or how many bytes were copied on a pointer dereference. The change in interpretation is only temporary and affects just the use with the cast. Consider this bit of C code:

```
{
   int *ptr;

   *(char *)ptr = 'a';
}
```

Assumptions: **ptr** address is in **R1**. Generated code:

```
   R2 = M[R1]        ; load value of ptr
   M[R2] =.1 97      ; deref ptr, copy just 1 char
```

The cast tells the compiler to treat the address as a **char *,** so when we dereference and assign to it, only one **char** is copied. If the value we were copying was larger than a **char** (say the value 1024), it would truncate and only copy the lower byte. Even though there is actually an **int** at that address, when handling this expression, the compiler treats the address as if there is only a **char** there.

Some typecasts are actually *type conversions*. A type conversion is required when the data needs to be converted from one representation to another, such as when changing an integer to floating point representation or vice versa.

```
   {
      int total, count;
      float average;

      average = ((float) total)/((float) count);
   }
```

Assumptions: **average** address is in **R1**. Generated code:

```
   R2 = M[R1 + 8]    ; load value of total
   R3 = ItoF R2      ; convert int to float
   R4 = M[R1 + 4]    ; load value of count
   R5 = ItoF R4      ; convert int to float
   R6 = R3/R5        ; divide (floating point version)
   M[R1] = R6        ; store in average
```

Sometimes a typecast doesn't change the generated code, it just sedates the compiler about incompatible types, such as assigning mismatched pointers, or assigning a pointer to an integer. Both pointers and integers are 4-byte integer-derived types and can be assigned to one another without any conversion, but the compiler (at least **gcc**) will

generate a warning unless you include a cast. So with or without the cast, the same machine code results, it's just a matter of quieting the compiler.

```
{
   char ch;
   int *s;

   s = (int *)&ch;
   *s = (int)s;
}
```

Assumptions: **s** address is in **R1**. Generated code:

```
R2 = R1 + 4       ; calculate address of ch
M[R1] = R2        ; store in s (no conversion required)
R3 = M[R1]        ; load value of s
M[R3] = R3        ; deref and store val there
```

Most often, a cast does affect the generated code, since the compiler will be treating the expression as a different type. You can do a lot of very questionable things with typecasts.

Behold the following:

```
{
   int i;

   ((struct binky *)i)->b = 'A';
}
```

Assumptions: **i** address is in **R1**. Generated code:

```
R2 = M[R1]           ; load value of i
M[R2 + 4] =.1 65     ; now treat like base address of struct!
```

What does this code actually do at runtime? Why would you ever want to do such a thing? The typecast is one of the reasons C is a fundamentally unsafe language. Even the best-laid walls of encapsulation cannot be strictly enforced with such a mechanism available. A malicious client can break through any barrier with just a simple cast. You can argue that this lack of restrictiveness contributes to C's expressiveness, and at times, it can be a valuable language feature, but it certainly comes with its downside.