COLORADO SCHOOL OF MINES
ELECTRICAL ENGINEERING & COMPUTER SCIENCE DEPARTMENT

CSCI-410
Elements of Computing Systems
Spring 2014

SUP-04

This is supplemental material to ECS Chapter 4 that explores HACK assembly language programming in more depth. In particular, we will look at translating high (or at least higher) level constructs into HACK assembly code. In addition, we will look at bit-banging and implementing shift and rotate instructions in software. Finally, we will look at a primitive, but common, way of implementing subroutines in assembly language.

Assignment Statements

Let's consider the following assignment statement:

```
a = b + c;
```

This can be implemented in HACK assembly with

```
@b
D = M
@c
D = D+M
@a
M = D
```

If one of the operands, say **c**, were a constant, say **K**, then this could be modified to

```
a = b + K;
```

This can be implemented in HACK assembly with

```
@b
D = M
@K
D = D+A
@a
M = D
```

In either case, it takes six instructions to implement this very simple statement.

But what if **a** and **b** happened to be the same variable? Then we could reduce the number of instructions by a third.

```
a = a + K;
```

This can be implemented in HACK assembly with

```
@K
D = A
@a
M = D+M
```

Finally, if the value of K happen to be 1, then we can shorten this even further and reduce the number of instructions by a total of two-thirds.

```
a = a + 1;
```

This can be implemented in HACK assembly with

```
@a
M = D+1
```

One of the primary strategies of both hardware and software design is to make the common case efficient. If we can shave just a small amount off of the time it takes to execute a commonly-performed operation, that will usually far overshadow the efficiency gained by making major reductions in the amount of time it takes to execute a rarely-performed operation.

It should not be surprising that adding (or subtracting) the constant 1 to the value of a variable is an extremely common operation in programming – we even have dedicated names for the operation: increment and decrement. Furthermore, these operations are often encountered inside loops (e.g., the loop counter) where performance gains (or penalties) are magnified. Thus, we would like our compilers to be able to recognize when these shorter sequences of code can be used.

But this is not a trivial task for a compiler; in essence, we are asking the compiler to recognize that it should generate fundamentally different code for the statements

```
a = 1 + a;
a = 1 + b;
a = 2 + a;
a = 2 + b;
a = b + c;
```

Today's compilers benefit from decades of research and experience on how to analyze source code and recognize when and where optimizations can be performed, as well as the astonishing improvements in computing resources, both speed and memory, that are available. In the early

days of computing (and still today in the case of many microcontrollers), compilers had to rely on cues provided by the programmer to know when and what optimizations were possible. This is why, for instance, the C programming language has three ways to write an increment operation:

```
a = a + 1;
a += 1;
a++;
```

While a really primitive compiler might generate the same general code for each that it would for

```
a = b + 1;
```

most compilers would generate the general code (six HACK instructions) for the first, the intermediate code (four HACK instructions) for the second, and the fast code (two HACK instructions) for the third.

Thus, a software developer that was sufficiently aware of the capabilities of both the hardware and the software tools at their disposal had the ability to significantly improve the performance of their code just by choosing which statement style to use in different situations. Not surprisingly, such developers were highly valued and very well paid. While today's optimizing compilers have diminished this somewhat, it is still true that performance critical applications still exist, particularly in the system-programming and embedded-programming worlds, and that developers that have detailed knowledge of "what's under the hood" – and know how to exploit it – still command premium salaries.

Program Flow Control

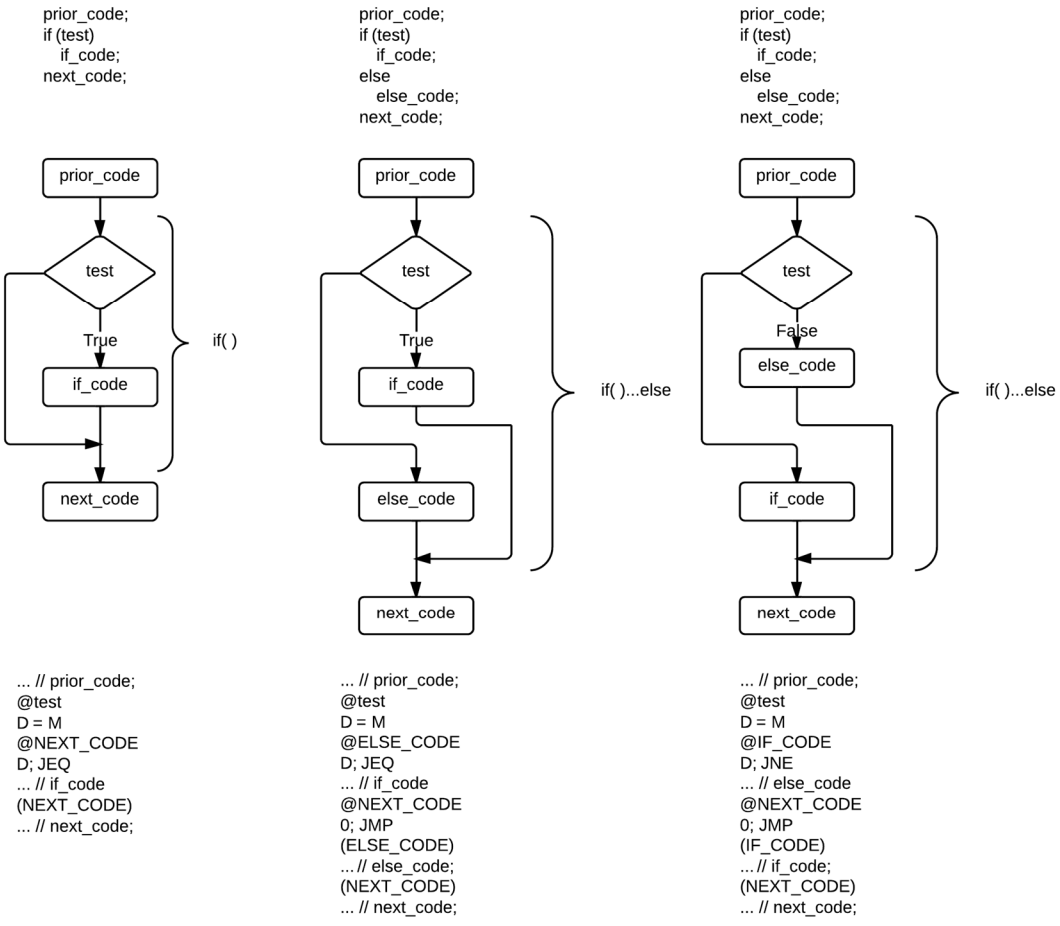
Programs are of strictly limited utility unless they have a means of altering the order in which instructions are executed in response to the data being processed. The most basic flow control is the unconditional jump, generally simply called a "jump" instruction. Because this instruction always transfers control, it does not give us the ability to achieve the goal of altering the order in which instructions are executed based on the data – at best it gives us the ability to create a loop (albeit an infinite one). To achieve true programmatic flow control, we need at least one instruction that implements a conditional jump, generally called a "branch". It should be noted that "jump" and "branch" are not standardized, so it is generally not safe to assume that a "jump" is unconditional or that a "branch" is – instead, if there is the possibility for any ambiguity, it is best to prefix either with the term "conditional" or "unconditional".

The capabilities of the processor to perform conditional branching need not be extensive – some processors have supported but a single instruction that merely skipped the next instruction if the value in the working register was zero. Any non-intrinsic capabilities, such as branching back to the beginning of a loop if the loop counter is less than some value, must be built up from whatever capability, however limited, the processor has.

Selection Constructs

The most basic (conditional) flow control is the if() structure in which some test is performed and a block of code is either executed or not depending on the outcome of the test. In nearly all computers, the test is Boolean, meaning that the result has one of two values, which we arbitrarily call "true" or "false". Having said that, nearly all computers have N-bit data paths, thus the value upon which the test is made has 2^N possible values. While it is certainly possible to design a processor that leverages this fact, the increase is hardware and software complexity doesn't justify it. Instead, the almost universal convention is that a value of exactly zero is deemed to be "false", while any non-zero value is deemed to be "true".

In the figure below, each column has a high-level language code template (patterned after the C programming language) followed by a flowchart that represents the logic and, below that, a HACK assembly language template corresponding to the flow chart. In each flow chart, the normal progression through the code is vertically downward. Any path that deviates from this is some type of branch – if a diamond is present the branch is conditional, as indicated by more than one path leaving the symbol, while if the path simply leaves the normal path is an unconditional jump.

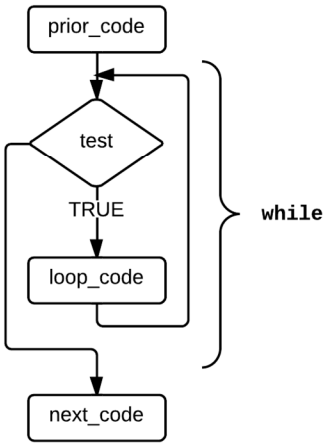


As you can see, there are two options available when implementing an if-else structure. The one on the left is the more intuitive as it more strongly matches the semantics of the logic as perceived by humans, but both are logically equivalent. However, they are not necessarily equivalent in terms of performance. If we assume that most of the time the "if" clause will be chosen, then we pay an extra penalty in the left if-else implementation because of the extra unconditional jump required to skip over the "else" clause. Of course, this assumes that there isn't a greater penalty for taking the conditional jump over not taking it; while many processors do have such a penalty, the HACK does not. Therefore, if we implement the if-else structure using the code on the right, we are likely to see a performance gain for the common case.

Loop Constructs

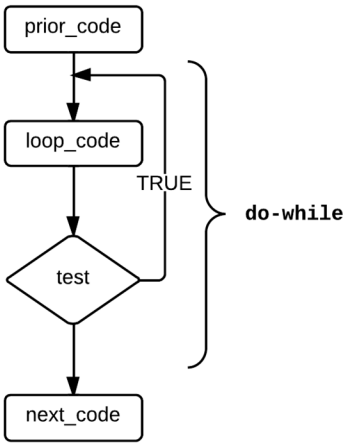
The two common looping structures are the while loop and the do loop, with the difference being that the while loop performs the test at the start of each pass through the loop, and thus has the option to not execute the loop at all, while the do loop has the test at the end of the loop, and thus guarantees that the loop will be executed at least once.

```
prior_code;
while (test)
  loop_code;
next_code;
```



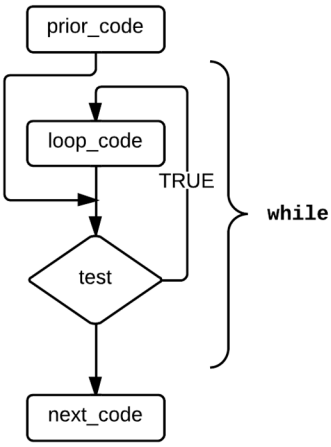

```
... // prior_code
(LOOP_CODE)
@test
D = M
@NEXT_CODE
D; JEQ
... // loop_code
@LOOP_CODE
0; JMP
(NEXT_CODE)
... // next_code
```

```
prior_code;
do
  loop_code;
while (test);
next_code;
```




```
... // prior_code
(LOOP_CODE)
... // loop_code
@test
D = M
@LOOP_CODE
D; JNE
... // next_code
```

```
prior_code;
while (test)
  loop_code;
next_code;
```




```
... // prior_code
@LOOP_TEST
0; JMP
(LOOP_CODE)
... // loop_code
(LOOP_TEST)
@test
D = M
@LOOP_CODE
D; JNE
... // next_code
```

Notice that in the high-level language, the while loop appears to be the simpler of the two constructs; yet examination of either the flow chart or the assembly code shows that this is deceptive and that the do loop actually has the cleaner implementation. This is not just aesthetic, either – in the (leftmost) while loop, both a conditional branch (as part of the test) and an unconditional jump (at the bottom of the loop) must be executed within each pass through the

loop. Both of these tasks are unavoidable, but the do loop is able to combine them into a single conditional branch since the test is at the bottom of the loop. However, we obtain this efficiency even in our while loop, as the rightmost implementation reveals, at the expense of one unconditional jump before the body of the loop. Not only do we gain the efficiency, but now our assembly language for a while loop and a do loop are essentially identical, with the only difference being where the "loop-entry point" is located.

Bit Banging

A common task, particularly in embedded systems, is working at the level of individual bits within a multi-bit value, also known as a bit vector. Few processors directly support bit-level operations, but virtually all support bitwise logical operations on multi-bit vectors. These can be utilized to effect bit-level operations by using a "mask" to single out the bits that are to be manipulated and safeguard the ones that aren't. This allows us to work with individual bits even though we have no choice but to always read and write entire multi-bit vectors.

The first thing that we have to be able to do is to generate a mask that picks off a particular bit of interest. This operation is often called MSK (or GEN), though there is really no widely accepted name for it. Our mask will need a single 1 in the bit position of interest with the rest of the bits being 0. If we number our bits starting with 0 for the lsb, then the value represented by the mask vector for bit n is simply 2^n.

$MSK(n) = 2^n$.

If we know the exact bit of interest at the time the program is written, then we can simply hardcode the mask as a constant value. On the HACK, this works fine for the low fifteen bits (bit 0 through bit 14), but it does not work for bit 15 since are A-type instruction always loads a zero for the msb. We can get around this by loading the value 32767, which will set all bits except bit 15, and then either adding 1 or inverting the entire vector (the latter being preferred in most cases).

If we will not know which bit is to be masked until run time, then we need a means of generating an arbitrary mask on the fly. A cursory examination should convince you that MSK(n) is equal to the value 1 left-shifted n places. On a processor that supports left-shift operations, this would map very nicely to the c statement.

```
mask = 1 << n;
```

However, the HACK does not support shift or rotate operations. But we can take note that MSK(0) is simply 1 and that MSK(n) = 2*MSK(n-1). Thus, given the mask for one bit position, we can generate the mask for the next higher bit position by doubling it, which on the HACK is easily accomplished by adding it to itself. In C, we therefore have an algorithm for MSK(n) which looks like the following:

```
uint16 MSK(uint16 n)
{
   uint16 mask, bit;

   for (mask = 1, bit = 0; bit < n; bit++)
      mask += mask;

   return mask;
}
```

A variant of this that results in tighter HACK assembly code would be

```
uint16 MSK(uint16 n)
{
   uint16 mask;

   for (mask = 1; n; n--)
      mask += mask;

   return mask;
}
```

A note regarding data types in the example C code: Whenever performing bit banging in C, it is wise to always use unsigned integers to prevent subtle logic errors that arise from the normal sign-extension rules that apply to signed integers when shift operations are performed. It is further generally good practice to explicitly control the width of those integers in order to make the code more portable.

Note that these algorithms will work without problem even if we want to set bit 15 since the only constant that must be loaded is the value 1.

Now that we can generate a mask, we can turn our attention to using that mask to perform bit-level operations. The two basic write operations are SET and CLR (these names are pretty widely accepted); with the former we wish to set a particular bit to a 1 while with the latter we wish to clear a particular bit to a 0.

For the SET operation, we us the fact that when any value is OR'ed with a 0, the result is the value while if any value is ORed with a 1, the result is a 1 regardless of the value. Therefore, we need our mask to contain a 1 in any bit locations that we wish to set and a 0 in any bit locations we wish to remain unaltered. Thus, our C language implementation for SET(value, bit) would look something like

```
uint16 SET(uint16 vector, uint16 bit)
{
   return vector | MSK(bit);
}
```

Conversely, for the CLR operation, we us the fact that when any bit is ANDed with a 1, the result is the bit's value while if any bit is ANDed with a 0, the result is a 0 regardless of the bit's value. Therefore, we need our mask to contain a 0 in any bit locations that we wish to clear and a 1 in any bit locations we wish to remain unaltered; this is merely the bitwise inverse of the mask for the desired bit.

```
uint16 CLR(uint16 vector, uint16 bit)
{
   return vector & ~MSK(bit);
}
```

The basic read operation is BTS, which stands for "Bit Test Set" and simply checks if a particular bit is set or cleared. We use the same concepts as with the SET and CLR, except somewhat in reverse. The goal is to generate a value that is exactly zero (false) if the bit being tested is clear and anything other than zero (true) if it is set. In order to get a result that is exactly zero, regardless of the state of the bits we are not interested in, we have to clear all the other bits in the result which is accomplished by using a mask that has a 0 in those positions and then ANDing them with the value being inspected. This then means that we will need to place a 1 in the position of interest which will mean that the result will be zero if that bit is 0 and the result will have a 1 in that position, making the overall result something other than zero, if that bit is a 1.

```
uint16 BTS(uint16 vector, uint16 bit)
{
   return vector & MSK(bit);
}
```

Shift and Rotate Operations

In a shift operation, all of the bits in a vector are moved a specificied number of places in a specified direction. There are three categories of operations: logical shifts, arithmetic shifts, and rotates.

Logical Shift: Any bits that are shifted past the end of the vector are lost. For those positions on the opposite end are filled with zeros. Hence, a shift in either direction by an amount equal to the number of bits in the vector will yield an all-zero vector.

Arithmetic Shift: Any bits that are shifted past the end of the vector are lost. For those positions on the opposite end, they are filled with zeros if the shift direction is to the left (making an arithmetic left shift the same as a logical left shift) but if the shift direction is to the right, then the vacated positions are filled with whatever the original state of the msb (the sign bit) was. This is referred to as "sign extension" and preserves the sign of the resulting value.

Rotate: Any bits that are shifted past the end of the vector are placed in the vacated bits on the opposite end. Hence, a shift in either direction by an amount equal to the number of bits in the vector will yield the original vector.

There are many variations on assembly language instructions to perform shifts and rotates with any given language typically supporting only a subset of them. In general, however, they fall into three groups: Those that shift/rotate only in a fixed direction and only by one place; those that shift/rotate only in a fixed direction but by an arbitrary number of places; and those that shift/rotate in an arbitrary direction by an arbitrary number of places. For our purposes, we will consider the following set of instructions:

	Logical Shift	Arithmetic Shift	Rotate
1 bit to left	**`sll`**	**`---`**	**`rotl`**
1 bit to right	**`slr`**	**`sar`**	**`rotr`**
n bits to the left	**`slln`**	**`---`**	**`rotln`**
n bits to the right	**`slrn`**	**`sarn`**	**`rotrn`**
n bits (to the right)	**`sln`**	**`san`**	**`rotn`**

While we could add these as instructions to our HACK assembly language instruction set (as macro instructions), doing so would significantly increase the complexity of the assembler. Instead, we will only consider the logic needed to accomplish each operation on the HACK platform as inline code.

First, we need to take inventory of what techniques are available for us to work with. First off, we really only have one instruction that lends itself to any shift or rotate operation and that is the ability to add a value to itself, thus doubling it and effecting a logic shift left by one bit position. Unfortunately, there is no corresponding way to effect a logical shift right, however we can note that a logical shift right by one position can be effected by performing a left rotate by fifteen positions and then forcing the most significant bit to be a zero.

This leads us to consider how we might implement a left rotate operation. This involves capturing the state of the leftmost bit (the sign bit) before effecting a logical shift left and then impressing that state onto the rightmost bit of the shifted result. This is actually fairly straightforward since we can detect the state of the sign bit given that it marks the value as being less than zero. We can use that to store either 0 or 1 in a temporary location. Then we can carry out the shift, which guarantees that the lsb will be zero, to which we can simply add the value stored in the temporary location.

To finish off our logical shift left, we need to force the leftmost bit to a zero, which we can accomplish by simply clearing it using a constant mask.

We now consider the issue of carrying out the sign extension necessary to effect an arithmetic shift right operation. One way to envision this is to perform a logical right shift by one, which ensures that the sign bit is cleared, and then copy the old sign bit, which is now bit 14, back into the sign bit location.

As you can see, performing some of these operations, while conceptually quite simple, can become very laborious on a platform that does not support even rudimentary shift or rotate operations at the hardware level. Since these are operations that are commonly used, particularly

in embedded systems, almost all commercial processors provide some level of hardware support even though it increases the complexity, size, and cost of the processor.

Primitive Subroutines

So far we have looked at the logic needed to carry out some commonly used operations, but we have not looked at how to turn the instructions that implement them from inline code segments into reusable functions or subroutines. One way to do this is to implement a generic framework for implementing and calling functions, which is exactly what will be done as part of the higher-level software tools for the HACK platform. However, these frameworks come at significant cost in terms of code size and performance, a cost which is frequently too expensive for resource-constrained embedded applications. Thus, it is quite common for application developers working at this level to adopt a much more primitive and restrictive approach in order to avoid this cost. The result is code that is, one the one hand, much more capable and scalable than code written without them would be, but also code that is not nearly as clean, simple, or safe as code written using a true function-calling framework.

In short, the primitive approach subroutines described here is a compromise that is best avoided, but not always avoidable. But more to the point, the mechanics involved in these primitive subroutines lay the foundation upon which a true function-calling framework is built, and are thus worth understanding from that perspective.

The two key issues that must be addressed in order to implement a subroutine is how data will be handled and how program flow to and from the subroutine will be handled. There are several approaches to both problems since the best approach is highly dependent not only on the capabilities and resources of the processor, but also on the nature of the subroutines and the application as a whole. For our purposes, we will adopt an approach in which each subroutine is assigned a small amount of RAM, called a scratch pad, that it and it alone may use. The calling code must therefore copy data into the subroutine's scratch pad, pass control to the subroutine, and then retrieve the results from the subroutine's scratch pad once control is returned. In order to be able to return control to the code that called it, one of the pieces of data that must be placed in the scratch pad is the address of the instruction to which the code should return; not surprisingly, this is known as the return address.

One restriction that results from this approach is that our subroutines are not "reentrant", meaning that only one instance of each subroutine can be executing at one time. Thus, recursion is not possible, either directly or indirectly. Additional challenges arise if the platform supports interrupts, which most do, but this is not a consideration for the HACK. As long as we minimize calling one subroutine from another and carefully manage the hierarchy of subroutines that do, we can avoid the demons that could otherwise be released.

Let's take as our example the subroutine **`sll;`** this function takes one argument, the value to be shifted, and returns one value, the shifted value. When possible, we will use the same memory for the return values as we do for the arguments, so we need one memory location for this purpose. In addition, we need a memory location for the return address. Since this will always be needed, we will place this location at the beginning of the subroutine's scratch pad.

Although traditionally all of the memory associated with a subroutine's scratchpad is in a single contiguous block, all that really matters is that it be reserved for the exclusive use of the subroutine. In the HACK assembly language, this can be done by simply assigning globally unique names to each of the memory locations in the scratchpad. We could do this using descriptive names such as **`sll.ra`** for the return address and **`sll.v`** for the vector value – and doing so would offer some readability benefits – but instead, we will adopt a more overall consistent convention that lends itself to automatic tool generation and suffixed the variables with a numerical index. We will then dictate that the return address is placed in index 0 while arguments and return values are placed in sequential indices starting with **`sll.1`**. While we give up the more descriptive names, we gain an independence from those names in that someone calling a particular subroutine only need know what order the arguments are in and not what the exact names by which they are known within the subroutine. We could use this same convention for any strictly local variables used by the subroutine, but there is little to be gained by that and so here we choose to claim the benefit of descriptive names.

Another issue that must be decided is what freedom our subroutine has to change the state of the machine in ways that will be visible once the subroutine returns. In processors with multiple registers, it is common to declare that some of them are for the use of the called subroutine and that it has no obligation to restore them to their original values while others are for the calling code and, if a subroutine uses them, it must restore them to their original values when finished. Because the HACK only has two registers, we will stipulate that the subroutine may alter either of them and that the calling code is responsible for saving any values it needs prior to transferring control.

Thus, our call to **`sll`** will look as follows, assuming that the value to be shifted is in the D register initially and that we want it to end up in the D register when we are finished.

```
@sll.1    // Place the argument into the scratch pad
M = D
@RA       // Place return address into the scratch pad
D = A
@sll.0
M = D
@SLL      // Transfer control to the subroutine
0; JMP
(RA)
@sll.1    // Fetch return value and place in D
D = M
…
(SLL)     // SHIFT LOGICAL LEFT BY 1
@sll.1    // Fetch first argument and place in D
D = M
M = D + M // Perform left-shift by one
@sll.0    // Return control back to calling code
0; JMP
```