CS 331
Spring 2016

Notes on x86-64 Assembly Language

Our compilers will translate BPL code into Intel x86 assembly code. Our lab machines all use 64-bit implementations of Linux, so we will use the 64-bit variant of x86 code, which some denote by x86-64, or even by x64. To some extent the 64-bit systems are backwards-compatible with 32-bit, but the same is not true of the assembly language. There are many, many books on 32-bit x86 architecture, but if you type one of their programs into our systems it will not assemble. Even the register names have changed in x64. The 64-bit world is more complicated. In x86-32, integers, floats and addresses were all 32 bits; the same assembly instructions worked for almost any data you would use. In the 64-bit world addresses are 64 bits and integers are 32 bits; even a simple move instruction has different versions for moving addresses and moving numbers. None of this is particularly difficult but you need to keep your head on straight when you are generating 64-bit assembly code. Fortunately, in this course we will deal with a very small subset of the whole assembly language – just a few addressing modes and about a dozen different instructions.

Once upon a time, before you were born, each chip had a dedicated assembly language that directly controlled the operation of the chip. Those days are long over. Assemblers now translate assembly code into microcode that controls the chip. Anything that can be easily translated to microcode can serve as an 'assembly language'. There are actually several different assemblers for the Intel chips – MASM is the Microsoft assembler, NASM is the popular "Netwide Assember", GAS is the Gnu assembler, and there are a number of others. Unfortunately, these don't all use the same syntax. GAS, which we will use, uses an old AT&T syntax that puts the source before the destination in an instruction:

```
movq %rsp, %rax
```

moves data from register RSP to register RAX. MASM and NASM use an Intel syntax that reverses this: to move data from RSP to RAX you would write something like

```
movq rax, rsp
```

(Intel syntax also eliminates the % as a register designator). This is another reason to be careful using assembly code from another source; you need to be sure which is the source and which is the destination in each instruction.

Note that our GAS assembler is case-insensitive. You can write an instruction as

```
MOVL $3, %EAX
```

or as

```
movl $3m %eax
```

Registers: There are 16 64-bit registers in our systems. Most of these are general purpose registers, though some have recommended uses. One caution is that we will tie into the C-library procedures for our I/O, and these make some assumptions about what is in various registers. The Intel standard recommends that specific registers be used for the first six arguments in function calls (a call that needs more than 6 arguments pushes the remainder on the stack). Except when we call C-routines we will ignore this; all of our internal function calls will work by pushing arguments onto the stack. This means that we have a lot of registers to play with.

Most of the 64-bit registers have names for their first 32-bit portions, for backwards compatibility with x86-32 systems. The 32-bit portions have names starting with 'e' – esp, eax, etc. The 64-bit versions have the same names, with the 'e' replaced by 'r': rsp, rax, and so forth. You need to connect the register with the type of data you are using. For example, if you want to move the integer 45 into register rax, you may either use

```
movl $45, %eax
```

or

```
movq $45, %rax
```

These do exactly the same thing, but you need to tie the opcode suffix to the size of the register you are using.

Note that in GAS,which we will use as our assembler, registers are denoted with a '%' character -- %rax, %rsp, etc. When you refer to those same registers in GDB you need to change the '%' to a '$'. Just to keep you on your toes....

Here is a list of the registers in the x86-64 architecture

Register	Lower half	Purpose
%rax	%eax	This is general purpose, but many people treat it as an "accumulator", where the results of calculations go. The Intel programming conventions call for return values to be placed in rax before returning.
%rsp	%esp, which you should never use	This is the stack pointer. It points to the top element currently on the stack. The stack grows towards smaller addresses, so a push operation decrements rsp. You can allocate local variables by decrementing rsp yourself, and pop the stack by incrementing rsp.
%rdi	%edi	This is general purpose. The Intel conventions call for the first argument for a function call to be passed in rdi. We will do this only when calling C routines, but you should be careful about trashing this register.
%rsi	%esi	The conventions call for the second argument to be passed in rsi.
%rdx	%edx	The third argument.
%rcx	%ecx	The fourth argument.
%r8	%r8d	The fifth argument.
%r9	%r9d	The sixth argument
%r12	%r12d	This is undesignated, and not used by the C-compiler.
%rbx	%ebx	These registers are designated "callee saved" in the conventions. This means that if you are using them you should save their prior values, and restore those values when you are done with them.
%rbp	%ebp	
%r10	%r10d	
%r13	%r13d	
%r14	%r14d	
%r15	%r15d	
%r11	%r11d	This is used for linking. I would avoid using it.

The "e" names for the lower 32-bits of 8 of these registers are the names of the x86-32 registers. Registers r8 through r15 were added in the x86-64 architecture, so their lower halves follow a different naming convention. There are other special purpose registers that you should not need to access. %rip is the instruction pointer, or address of the next instruction to be executed. %rflags is the machine status word

Addressing Modes: The GAS assembler uses an AT&T format in which instructions are written

```
opcode  S, D
```

where S is the "source" and D is the "destination" for the instruction. A few instructions take only one or no operands; there are no 3-operand instructions. The operands themselves have a variety of possible formats. We will use only the following:

Register mode: The operand is the contents of the specified register. For example,

```
movq %rsp, %rax
```

uses register mode for both the source and destination operands. The effect of this instruction is to move the contents of register rsp to register rax.

Indirect mode: This gives a register and an offset in the format offset(%register). The operand is the contents of the memory location given by adding the offset to the register value. For example,

```
movl  %eax, 8(%rsp)
```

moves the 32 bits of register eax to the stack location given by 8-bytes below the current top of the stack.

Immediate mode: This specifies a number as the actual operand. The number needs to be preceeded by a '$':

```
movl  $23, %eax
```

puts the value 23 into the 32-bit register eax. You can do this with labels as well. For example, to call the C-routine printf we will use an instruction

```
movq  $S1, %rdi
```

where S1 is a label on the memory location that contains the desired formatting string. This puts the address of S1 into register rdi. When used with labels rather than numbers I call this "absolute mode".

Direct mode: I am not sure this is a standard name. If you have a label on a memory location where data is stored and you want to do something with the contents of that memory you can use the label without a '$' prefix:

```
movl  X, %eax
```

moves the 32 bits at label X into register eax.

Instructions: We will use a surprisingly small subset of the full x86 instruction set. Most instructions have a variety of possible suffixes, the 'l' suffix indicates 32-bit data, while the 'q' suffix indicates 64 bits. We will use only 3 categories of instructions – those that move data around, those that affect the control flow, and those that do arithmetic

Data Instructions:

Move moves the source data into the destination

`movl S, D` moves 32 bits; so the source needs to be 32 bits and the destination needs to be either a memory location or a 32-bit register.

`movq S, D` moves 64 bits

Load Effective Address puts the address of the source into the destination

`leaq 8(%rsp), %rax` puts the address of the first word below the top of the stack into rax. You could achieve the same effect in two instructions by putting the stack pointer into rax and adding 8, so you never really need to use this.

Clear loads 0 into the destination

`clrl D` puts 0 into the 32-bit destination

`clrq D` puts 0 into the 64-bit destination.

Push pushes the source operand onto the stack

`push S` decrements rsp by 8 bytes and puts the contents of S at this location on the stack. rsp always points at the top value on the stack.

Pop puts the top of the stack into the destination and increments rsp by 8 bytes.

`pop %rax` pops the stack into rax. If you just want to get things off the stack and not put them somewhere, you can achieve this by adding the appropriate value to rsp.

Control Flow Instructions:

Jump This puts the destination into the instruction pointer rip; the destination is almost always as label, as in

```
jmp L1
```

Compare compares the destination and source operands. It is usually followed by one of the conditional jumps. We will only use it to compare integers, so this form is

`cmpl S, D`

Conditional jumps: `je, jne, jl, jle, jg, jge, jz` These all take a destination, usually a label, as their only operand. The condition is the result of the previous instruction, which is usually cmpl. Unfortunately, the AT&T syntax GAS uses conflicts with the instruction names. The instructions

```
cmpl $8, %eax
jle  L2
```

result in control jumping to label L2 if eax <= 8; most people reading the instruction would expect the reverse. The reason for this dissonance is that that AT&T syntax reverses the order of the operands; the same statements written in Intel syntax would branch of 8 <= rax. You need to remember this when you implement the comparison operators. Note that the `jz` instruction jumps if the result of the previous instruction was zero – it doesn't require a cmpl instruction.

Call pushes rip (the address of the next instruction, also known as the return address) onto the stack and jumps to the start of the function being called, which is almost always marked with a label. For example

`call F` pushes the address of the next instruction onto the stack and jumps to label F. Note that the instruction itself does nothing with arguments and nothing with local variables. Note also that function F should assume when it starts that the proper return address is at the top of the stack. Anything F pushes onto the stack needs to be popped back off before it executes a return instruction.

Return pops the stack into the instruction pointer rip. That is all it does. A frequent cause of your compiled code crashing is not handling the stack properly, so when a return statement is executed there is something other than the return address at the top of the stack. Some malware tries to get control of a machine by pushing the address of a malicious function onto the stack just before a return. The return instruction has no operands; it is just

```
ret
```

Arithmetic Instructions: You would think these would be simple.

Addition, **Subtraction**. In these the source operand is added or subtracted to the destination. Use the 'l' suffix for integer arithmetic, the 'q' suffix for addresses.

`addl $8, %eax` Add 8 to the integer in eax

`addq $8, %rsp` Add 8 to the stack pointer, effectively popping the stack

Multiplication. The MUL instruction produces 64-bit output and doesn't work the way you would (or at least the way I would) expect. imul is32-bit integer multiplication operator. For example

`imul 0(%rsp), %eax` multiplies the value at the top of the stack times the value in register eax, and leaves the result in eax.

Division. This is more complex. The basic division operation is set up to divide 128 bits, stored in two registers, by 64 bits stored in one. If the dividend is negative we need to "sign-extend" it to fill up the rest of its 128 bits with copies of the sign bit (0's for positive dividends, 1's for negative dividends). Here is a sequence of steps that makes it work:

a) Put the divisor into ebp.
b) Put the dividend (the number being divided into) into eax.
c) Do a cltq instruction (with no operands) to sign-extend it to all of rax. cltq stands for "convert long to quad" – a "longword"is 32 bits, a quadword" is 64.
d) Do a cqto instruction (with no operands) to sign-extend it to rdx. cqto stands for "convert quadword to octword."
e) Do a idivl instruction, whose only argument is the divisor ebp. The quotient is put into eax, and the remainder (for a mod or % operation) is put into edx. For example,

```
movl $7, %ebp
movl $23, %eax
cltq
cqto
idivl %ebp
```

the result leaves the quotient, 3 in eax and the remainder, 2 in edx

Assembler Directives: There are a few directives that tell the assembler how to deal with the text of an assembly language program. Here are ones we will use, in the order in which you will probably use them:

.comm Symbol bytes, alignment as in `.comm X, 40, 32` This allocates a named chunk of read/write memory using the given number of bytes. We use this for global variables and arrays.

.section .rodata The directives that come after this are put in a "readonly" section, where C expects to find string constants. You will at least need to define strings here for working with the scanf and printf routines. To define a string, use

Symbol: .string <literal string in double quotes >

For example, you might have at the top of your code the directives

```
.section .rodata
.LC0: .string "%d "
.LC1: .string ">>> "
```

.text This marks the start of the executable assembler code.

.globl Symbol This indicates to the linker that the given symbol has global scope in the program. We link to a C run-time environment that expects a top-level function called main( ). BPL also requires a function called main. To indicate that this function is the global main( ) expected by the run-time environment use the directive

```
.globl main
```

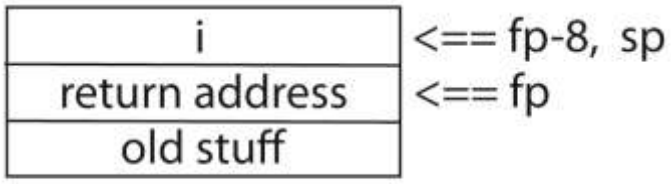
You can give this at the start of the .text section; the globl directive does not need to be at the point where the main function is defined.

Example: The following program is an assembly language version of

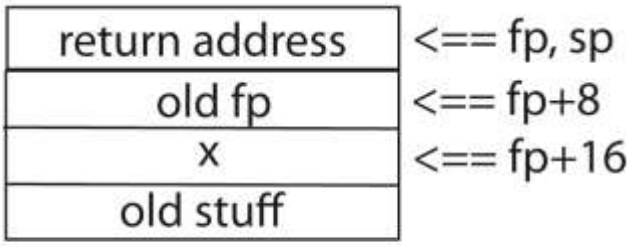
```
int f(int x) {
        return 2*x;
}

void main(  void ) {
        int i;
        i = 0;
        while (i < 10 ) {
                write( i );
                write( f(i) );
                writeln( );
                i = i + 1;
        }
}
```

This is hand-generated code, not code generated by my BPL compiler (which is much less efficient). You should be able to follow the code instruction by instruction. Note that this uses %rsp as the stack pointer and %rbx as the frame pointer. The following diagrams show the stack during the calls to main( ) and f( ):



The frame for main( )



The frame for f(x)

.

```
.section   .rodata
.WriteIntString: .string "%d "
.WriteInString: .string "\n"
.text
.globl main

f:
        movq %rsp, %rbx          # set up the frame pointer
        movq 16(%rbx), %rax      # argument value
        imul $2, %eax            # performing multiplication
        ret                      # return from the function
main:
        movq %rsp, %rbx          # set up the frame pointer
        sub $8, %rsp             # allocate local variable i
        movl $0, %eax            # putting value into ac
        movl %eax, -8(%rbx)      # assign to i
.L0:
        cmpl $10, -8(%rbx)       # compare i and 10
        jge .L1                  # if i >= 10 leave the loop
        movl -8(%rbx), %esi      # value to print  (arg2 for the call)
        movq $.WriteIntString, %rdi
        movl $0, %eax            # clear the return value
        call printf              # call the C-lib printf function
        push -8(%rbx)            # pushing argument for the call to f
        push %rbx                # pushing the frame pointer
        call f                   # calling the function
        pop %rbx                 # retrieving the frame pointer
        add $8, %rsp             # removing args from the stack
        movl %eax, %esi          # value to print (arg2 for the call)
        movq $.WriteIntString, %rdi
        movl $0, %eax            # clear the return value
        call printf              # call the C-lib printf function
        movq $.WriteInString, %rdi
        movl $0, %eax            # clear the return value
        call printf              # call the C-lib printf function
        movl -8(%rbx), %eax      # value of i
        addl $1, %eax            # performing addition
        movl %eax, -8(%rbx)      # assign
        jmp .L0                  # WHILE: jump back to top
.L1:
         add $8, %rsp            # deallocate local variables
         ret                     # return from the function
```