CS360 Lecture notes

CS360 Lecture notes
Assembler Lecture #1: Introduction; Global and local variables

- James S. Plank
- Directory: **/home/jplank/cs360/notes/Assembler1**
- Start Date: Sometime in the 1990's
- Latest revision: Wed Feb 14 15:26:21 EST 2018
- Git Hash: 96d4718adaafb536f67d2e708e8ff0af3ca2c1aa

This is the beginning lecture on computer organization & stack frames.

My goal is to teach you how assembly code is generated by a compiler to work on a standard processor. I don't do this in a precise manner, as would be done in a compiler course, but in a manner that is intended to be more intuitive.

The assembly code is a made-up RISC assembly code that works on a fictitious machine that has 4-byte pointers and no floating point. I have written a visual assembler for it so that you can see it in action.

Registers

We're going to assume a general computer architecture, which differs slightly from most machines, but is exemplary of almost all uniprocessors.

We'll assume that our machine has 8 general-purpose registers in the CPU. All are 4 bytes and can be read or written by the user. The first five are named **r0**, **r1**, **r2**, **r3**, **r4**. The last three registers are special:

- The sixth is named **sp** and is called the "stack pointer."
- The seventh is named **fp** and is called the "frame pointer."
- The eighth is named **pc** and is called the "program counter."

Additionally, the computer has three read-only registers, which always contain the same values:

- **g0**, whose value is always zero.
- **g1**, whose value is always one.
- **gm1**, whose value is always negative one.

Finally, the computer also has two special registers that the user cannot access directly:

- **IR** – The instruction register. It holds the instruction currently being executed.
- **CSR** – The control status register. It contains information pertaining to the execution of the current and previous instructions.

The instruction cycle

The computer's operation consists of running instructions repetitively. This is known as the instruction cycle. The instruction cycle consists of 4 general phases:

1. Decode instruction (in **IR**)
2. Execute instruction
3. Determine next instruction and update the **pc** accordingly
4. Load next instruction into the **IR**

What is an instruction? Like everything else, it's a sequence of 0's and 1's. We're going to assume that all of our instructions are 32 bits (although that's a naive assumption, it will work for our purposes). Instructions are stored as part of a program's memory, and the instruction that is pointed to by the **pc** register is the one that gets loaded into the **IR** for execution.

In other words, if the **pc** contains the value `0x2040`, then the **IR** is executing the instruction contained in the 4 bytes starting at memory address `0x2040`.

Assembly code is a readable encoding of instructions. A program called an *assembler* converts assembly code into the proper 0's and 1's that compose the program. If you call `gcc` with the `-S` flag, it will produce a `.s` file containing the assembler for that C program. Without the `-S` flag, it produces the instructions directly.

Instructions

1. Memory <-> Register instructions:

```
    ld mem -> %reg        Load the value of the register from memory.

    st %reg -> mem        Store the value of the register into memory.
```

There are a few ways to address memory:

```
    st %r0 -> i         Store the value of register r0 into the memory
    location of global variable i.

st %r0 -> [r1]      Treat the value of register r1 as a pointer
    to a memory location, and store the
    value of r0 in that memory location.

st %r0 -> [fp+4]    Treat the value of the frame pointer as a
    pointer to a memory location, and store
    the value of r0 in the memory location 4
    bytes after that location.  You can use
    any value, positive or negative -- the
    value must be a multiple of four.
    However, you cannot use a register (i.e.
    you can't do st %r0 -> [fp+r2]).  This
    only works with the frame pointer.  It
    does not work with any other register.

st %r0 -> [sp]--    Treat the value of register sp as a
    pointer to a memory location, store the
    value of r0 into that memory location, and then
    subtract 4 to the value of sp.

st %r0 -> ++[sp]    Treat the value of register sp as a
    pointer to a memory location. First, add 4 to
    that value, then store the
    value of r0 into that memory location.
```

2. Register <-> Register instructions:

```
mov %reg -> %reg    Copy a register's value to another
mov #val -> %reg    register, or set its value to a constant.
```

All arithmetic goes from register to register:

```
add %reg1, %reg2 -> %reg3   Add reg1 & reg2 and put the sum in reg3.
sub %reg1, %reg2 -> %reg3   Subtract reg2 from reg1.
mul %reg1, %reg2 -> %reg3   Multiply reg1 & reg2.
idiv %reg1, %reg2 -> %reg3  Do integer division of reg2 into reg1.
imod %reg1, %reg2 -> %reg3  Do reg1 mod reg2.
```

There are two special instructions that let you perform addition and subtraction on the stack pointer:

```
push %reg           This subtracts the value of %reg or #val
push #val           from the stack pointer.

pop %reg            This adds the value of %reg or #val
pop #val            to the stack pointer.
```

3. Control instructions

```
jsr a                   Call the subroutine starting at instruction a.
ret                   Return from a subroutine.
```

There are also "compare" and "branch" instructions, which is how you implement for and if statements, but I won't go over them yet.

Finally, there are also "directives" which are not really code, but specify that memory must be allocated for variables. In this assembler, this is just one such directive:

```
.globl i           Allocate 4 bytes in the globals segment
for the variable i.
```

The program counter points to where the instruction register must go to load its value. On normal instructions, the **pc** is incremented by 4 so that the next instruction can be loaded. On control instructions, the **pc** gets a new value, allowing the machine to call subroutines, perform "if-then" statements, etc.

The address space

Each program's view of its memory is called an "address space". Typically, an address space is broken up into 4 parts: The *code*, *globals*, *heap*, and *stack*. The code holds nothing but instructions. The globals is where global variables are stored, and the heap is where malloc'd storage lives. The stack is for temporary storage, like local variables and arguments for procedures.

Generally, a process treats memory like a huge array of bytes; however, the bytes are organized logically into units of 4 bytes each, as that is the size of registers. We assume that this memory is of size `0x80000000`. The code starts at address `0x1000`. The globals follow the code, starting in the first address that is a multiple of 4096 (`0x1000`). On a real machine, the heap would follow the code, or start at some other address which is a multiple of 4096. With **jassem**, there is no heap.

As a program executes, the stack will grow and shrink, but the code and globals stay the same size. The stack grows from back to front, starting at address `0x80000000` and growing toward lower memory addresses. In between the globals and stack is unused memory:

```
The programs' address space:
|--------------------------|<------- 0x00000000
|  No access (Seg faults)  |
|--------------------------|<------- 0x00001000
|                          |
|          code            |
|                          |
|--------------------------|
|                          |
|         globals          |
|                          |
|--------------------------|
          ....
       Unused memory
          ....
|                          |
|      (grows up)          |
|^^^^^^^^^^^^^^^^^^^^^^^^^^|
|                          |
|         stack            |
|                          |
|--------------------------|<------- 0x80000000
```

Simple compiled code.

The C compiler takes C code, and translates it into instructions. What we're doing in this and the following lectures is seeing how this translation works. The assembler code produced by the translation consists of machine instructions and directives. The translation is very logical. For example, the following code:

```
int i;
int j;

int main()
{
    i = 3;
    j = 2;
    j = i + j;
}             /* I am intentionally not returning anything. */
```

Will compile into the following assembly code:

```
    .globl i                    // Allocate i and j as global variables.
    .globl j
main:
    mov #3 -> %r0               // i = 3
    st  %r0 -> i
    mov #2  -> %r0              // j = 2
    st  %r0 -> j
    ld  i   -> %r0              // j = i + j
    ld  j   -> %r1
    add %r0,%r1  -> %r1
    st  %r1 -> j
    ret
```

This code is pretty straightforward. Each instruction in C has a corresponding set of instructions in assembler. Unless your compiler is smart, it will produce inefficient code. For example, you can probably see that:

```
        .globl i
        .globl j
    main:
        mov #3  -> %r0
        mov #2  -> %r1
        add %r0,%r1  -> %r1
        st  %r1 -> j
        st  %r0 -> i
        ret
```

This would work just as well and has fewer instructions. If you call `gcc` with the `-O` flag, it will attempt to optimize your code so that it has fewer instructions. However, normally, `gcc` simply produces straightforward, unoptimized code. In this class (I'll repeat this over and over), we are going to produce unoptimized code, which means that every C statement is translated to assembly code independently. No assumptions are made from one statement to the next. We leave compiler optimization to another course (like a compiler course).

Now, suppose instead that we have the following code to run:

```
int main()
{
    int i, j;

    i = 3;
    j = 2;
    j = i + j;
}
```

Since **i** and **j** are local variables, they must come from temporary storage: The stack. How does the stack work? It is governed by the **sp** and **fp** registers. The **sp** and **fp** designate what is known as a "frame" on the stack. The **fp** points to the bottom of the frame, and the **sp** points to the top. All memory locations above (less than or equal to) the stack pointer are considered unused. Thus, we can get new temporary memory by decrementing the **sp**, thus putting memory locations into the current stack frame.

For example, when a procedure is first called, these two registers point to the same place in the stack. The frame is considered empty.

```
....
|----------------|
|     unused     |
|----------------|
|     unused     |  <------------- sp, fp
|----------------|
|      used      |
|----------------|
|      used      |
|----------------|
       ....
|----------------|
|      used      |
|----------------|
                        <---------- Location 0x80000000
```

To allocate room for the two local variables **i** and **j**, we decrement the stack pointer by 8. This allocates two 4-byte quantities in the current stack frame: By convention, we'll call the lower one **j**, and the upper one **i**. This is something that the compiler defines. We could just as easily have called the lower one **i**, and the upper one **j**:

```
....
|----------------|
|     unused     |  <------------- sp
|----------------|
|       i        |
|----------------|
|       j        |  <------------- fp
|----------------|
|      used      |
|----------------|
|      used      |
|----------------|
       ....
|----------------|
|      used      |
|----------------|
                        <---------- Location 0x80000000
```

Now, the code for **main()** is just like before, only instead of accessing **i** and **j** as global variables, we access them as offsets to the frame pointer.

```
main:
push #8                  // This allocates i and j
mov #3   -> %r0
st  %r0 -> [fp-4]        // Set i to 3
mov #2  -> %r0
st  %r0 -> [fp]          // Set j to 2
ld  [fp-4]  -> %r0
ld  [fp]    -> %r1
add %r0,%r1 -> %r1       // Add i and j and put the result
st  %r1 -> [fp]          // back into j
ret
```

Let's look at what happens when **main()** is executed:

```
Stack
|----------------|
|                |                                registers
|                |                            |-----------------|
|                |                            |                 | r0
|                |                            |                 | r1
|     .....      |                            |                 | r2
|     unused     |                            |                 | r3
|     unused     |                            |                 | r4
|     unused     |        /-----------        |                 | sp
|     unused     |  <------------------       |                 | fp
|      used      |                            |    main         | pc
|      ....      |                            |-----------------|
|--------------- |
```

Note that the **fp** and **sp** point to the base of the empty stack frame. The **pc** points to the beginning of the main routine. This is the instruction "**push #8**". When this is done executing, we have the following:

```
Stack                               registers
|----------------|                           |-----------------|
|                |                           |                 | r0
|                |                           |                 | r1
|     .....      |                           |                 | r2
|     unused     |                           |                 | r3
|     unused     |  <---------------\        |                 | r4
|       i        |                   \-      |                 | sp
|       j        |  <------------------      |                 | fp
|      used      |                           |    main + 4     | pc
|      ....      |                           |-----------------|
|--------------- |
```

Space has been allocated on the current stack frame for **i** and **j**, and the **pc** has been incremented. It now points to the instruction "**mov #3 -¿ %r0**". This puts the machine into the following state:

```
Stack                               registers
|----------------|                           |-----------------|
|                |                           |       3         | r0
|                |                           |                 | r1
|     .....      |                           |                 | r2
|     unused     |                           |                 | r3
|     unused     |  <---------------\        |                 | r4
|       i        |                   \-      |                 | sp
|       j        |  <------------------      |                 | fp
|      used      |                           |    main + 8     | pc
|      ....      |                           |-----------------|
|--------------- |
```

Now, the **pc** points to "**st %r0 -¿ [fp-4]**". When this is done, the location for **i** is set to the value 1:

```
    Stack                              registers
    |----------------|                     |-----------------|
    |                |                     |       3         | r0
    |                |                     |                 | r1
    |    .....       |                     |                 | r2
    |    unused      |                     |                 | r3
    |    unused      | <---------------\   |                 | r4
    |       i=3      |                  \- |                 | sp
    |       j        | <------------------ |                 | fp
    |     used       |                     |    main + 12    | pc
    |     ....       |                     |-----------------|
    |--------------- |
```

After the next two instructions, the state of the machine will be:

```
    mov #2   -> %r0
    st  %r0 -> [fp]
```

```
  Stack                              registers
|----------------|                     |-----------------|
|                |                     |       2         | r0
|                |                     |                 | r1
|    .....       |                     |                 | r2
|    unused      |                     |                 | r3
|    unused      | <---------------\   |                 | r4
|       i=3      |                  \- |                 | sp
|       j=2      | <------------------ |                 | fp
|     used       |                     |    main + 20    | pc
|     ....       |                     |-----------------|
|--------------- |
```

Finally, the last 4 instructions do:

```
    ld  [fp-4]  -> %r0
    ld  [fp]    -> %r1
    add %r0,%r1 -> %r1
    st  %r1     -> [fp]
```

```
  Stack                              registers
|----------------|                     |-----------------|
|                |                     |       2         | r0
|                |                     |       3         | r1
|    .....       |                     |                 | r2
|    unused      |                     |                 | r3
|    unused      | <---------------\   |                 | r4
|       i=3      |                  \- |                 | sp
|       j=3      | <------------------ |                 | fp
|     used       |                     |    main + 36    | pc
|     ....       |                     |-----------------|
|--------------- |
```

Jassem – The Visual Assembler

To help you understand assembler, I have written a simple visual assembler that lets you load assembly code programs and step through them. It has been written in the graphical scripting language **tcl/tk**.

You can get this file from this directory, in **jassem.tcl**.

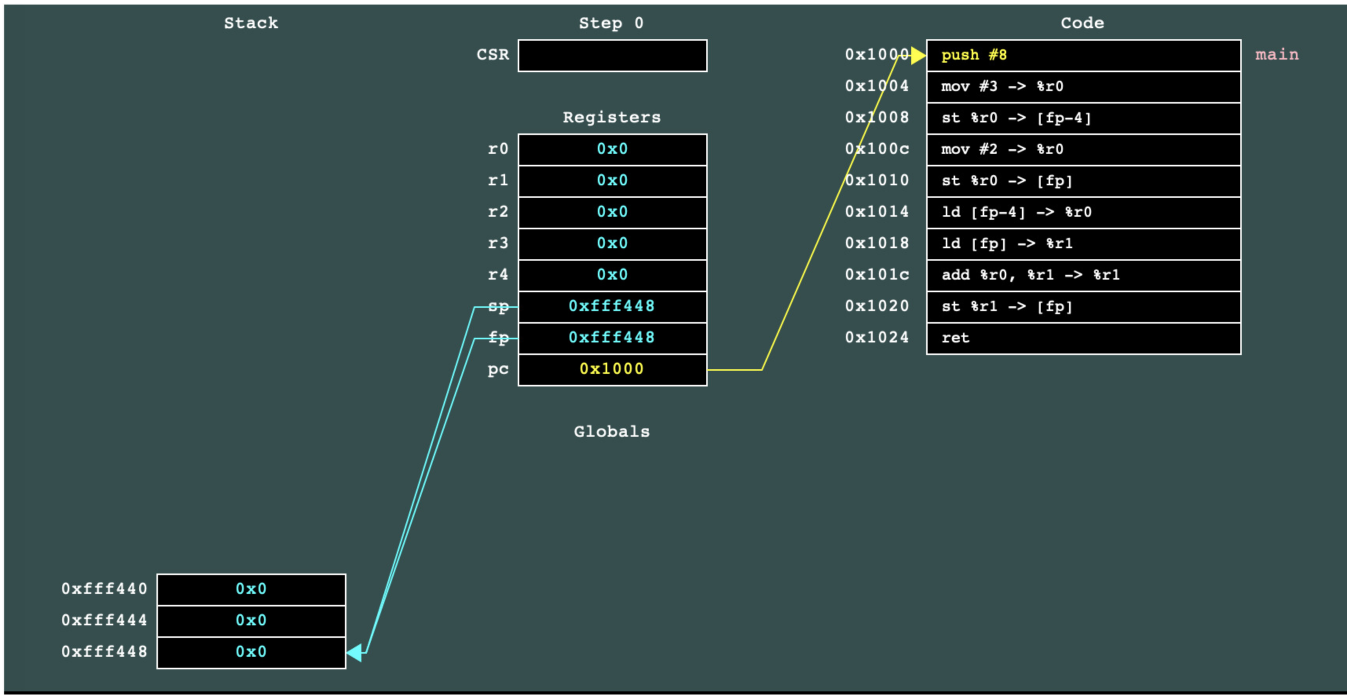
The nice thing about tcl/tk is that it works on Unix, Windows, and Macintosh. To use **jassem.tcl** on our machines, simply run:

```
UNIX> wish ~jplank/cs360/bin/jassem.tcl [filename]
```

Wish should be installed on all of our machines.

To use **jassem.tcl** on a Windows or Macintosh machine, you will need to install tcl/tk. This is free – get the code from **www.scriptics.com**.

In running **jassem.tcl**, the first thing you do is load a program, such as **p1-g.jas** (the program above that adds global variables) or **p1.jas** (the program above that adds local variables). You should see a picture of the system – stack, registers, globals and code, as below:



```
CSR

Registers
r0   0x0
r1   0x0
r2   0x0
r3   0x0
r4   0x0
sp   0xfff448
fp   0xfff448
pc   0x1000
```

```
0x1000  push #8
0x1004  mov #3 -> %r0
0x1008  st %r0 -> [fp-4]
0x100c  mov #2 -> %r0
0x1010  st %r0 -> [fp]
0x1014  ld [fp-4] -> %r0
0x1018  ld [fp] -> %r1
0x101c  add %r0, %r1 -> %r1
0x1020  st %r1 -> [fp]
0x1024  ret
```

```
0xfff440  0x0
0xfff444  0x0
0xfff448  0x0
```

Now you can step through the program, looking at everything as you go. Make sure you understand each step as you go through it. This is a very helpful tool.

In **jassem**, the code starts at 0x1000, and global variables, if they exist, start at the next multiple of 0x1000. There is no heap. The stack ends at 0xfff44c. That's pretty arbitrary, but such is life.

Addendum

These questions came from a student in cs360 many many years ago – since I imagine many students might share these questions, I am broadcasting the answers to everyone.

```
>1. Does the term uniprocessor mean that we have only one CPU, or is there
>   something else I should understand?
```

That is correct – one CPU (as opposed to a parallel processor that has many CPUs)

```
>2. In the register <--> memory operation
>                     st %r0 --> [r1]
>   the notes said "... treat the value of register r1 as a pointer to a memory
>   location ...". Except for the pc, fp and the sp do we ever know where in
>   memory a register is pointing ?
```

You can assume that the pc points to the code, and the fp/sp both point to memory in the stack. Even these assumptions can be violated in some systems if you are doing complex stuff (I won't go into it). Otherwise, you cannot assume that a register is pointing to a specific memory segment. r1's pointer can point to the code, globals, heap or stack.

```
>3. Does each process assume that the address space is 0x80000000 ? If this is
>   the number of bytes it looks huge to me. I got exactly 2048Mb. The hydras,
>   for example, have only 96Mb of RAM.
>   I got 2048 by doing
>                       8*(16^7)/(1024^2).
>   Am I making any wrong assumptions ? Things don't seem right.
```

Yes, the process assumes that memory is an array of 2 GB. However, it won't use all 2 GB. In particular, the addresses between the stack and heap are unused, and they compose the bulk of the address space. Even though a processor may have much less than 2 GB in RAM, the system is set up to look as though each process can access 2 GB. This is called "virtual memory", and is something that you'll learn about in CS361. In a few weeks, we'll see how your interface to memory is limited. In particular, usually your code and globals segment is smaller than a megabyte. On my machine, the OS does not allow the stack to grow larger than 8M or so, (type "limit" and look at "stacksize") and it does not allow the heap to grow too much larger than 96MB. If you don't believe me – try it:

The program **test1.c** tests the heap:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char **argv)
{
    char *s;
    long sz;

    if (argc != 2 || sscanf(argv[1], "%ld", &sz) == 0) {
    fprintf(stderr, "usage: test1 bytes\n");
    exit(1);
    }

    s = (char *) malloc(sz);

    if (s == NULL) { perror("malloc"); exit(1); }
    printf("malloc %ld worked\n", sz);
    return 0;
}
```

In 2018, you had to allocate quite a bit of memory to have the hydra's fail:

```
UNIX> gcc test1.c
UNIX> a.out 6000000000
malloc 6000000000 worked
UNIX> a.out 7000000000
malloc: Cannot allocate memory
UNIX>
```

My Raspberry Pi fails quicker:

```
pi@raspberrypi:~$ gcc test1.c
pi@raspberrypi:~$ ./a.out 400000000
malloc 400000000 worked
pi@raspberrypi:~$ ./a.out 500000000
malloc: Cannot allocate memory
pi@raspberrypi:~$
```

This program (**test2.c**) tests the stack:

```
#include <stdio.h>

int main()
{
    char s[9000000];

    printf("s = 0x%x\n", s);
    printf("%d\n", *s);
    return 0;
}
```

As you can see below, our stacks are (as of 2018) limited to 8 MB, which seems like plenty of memory to me. So, when **test2.c** tries to allocate 9,000,000 bytes on the stack, you get a seg fault:

```
UNIX> limit | grep stack
stacksize    8192 kbytes
UNIX> gcc test2.c
UNIX> ./a.out
Segmentation fault
UNIX>
```

If we instead only allocate 8,000,000 bytes, it succeeds:

```
UNIX> sed 's/90/80/' test2.c > test3.c
UNIX> gcc test3.c
UNIX> a.out
s = 0xaf67d5d0
0
UNIX> rm test3.c a.out
UNIX>
```
