CSC258: Computer Organization

Functions and the Compiler Tool Chain

A Note about This Week's Quiz

- There were very few exercises this week, except for writing substantial pieces of code.

- Instead, I provided questions in the reading guide. I would treat these as the set of questions that should be prepared for the quizzes.

Discussion: Reading "Quiz"

1. What are the names of the tools in the compiler toolchain? (And what is the rough responsibility of each?)

2. What does the *jal* (jump and link) instruction do?

3. List the expectations we, as programmers, have of functions. Here's one …
   a. After a function executes, control returns to the instruction *after* the function call at the call site.
   b. …

Discussion: Reading "Quiz"

1. What are the names of the tools in the compiler toolchain? (And what is the rough responsibility of each?)

2. What does the *jal* (jump and link) instruction do?

3. List the expectations we, as programmers, have of functions. Here's one …
   a. After a function executes, control returns to the instruction *after* the function call at the call site.
   b. …

The Compiler Toolchain

Questions to Consider

- What are the components of the compiler toolchain? What are their inputs and outputs?
- Why does the assembler take two passes?
- Why does the linker need to be able to "patch" addresses?
- Why don't we do any type checking when combining files (with the linker)?

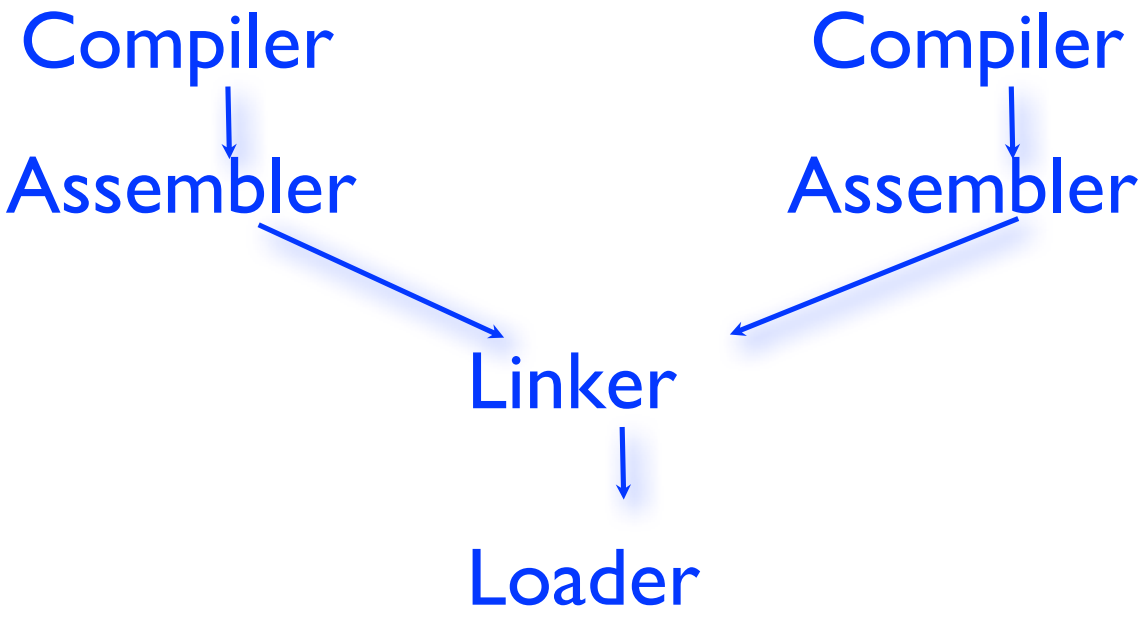
Disclaimer

- These slides present an idealized compiler toolchain.

- In the “real world”, the responsibilities and distinctions between components are blurred.
  - For example, the linker and loader may be the same application!
  - The compiler and assembler may also be combined.

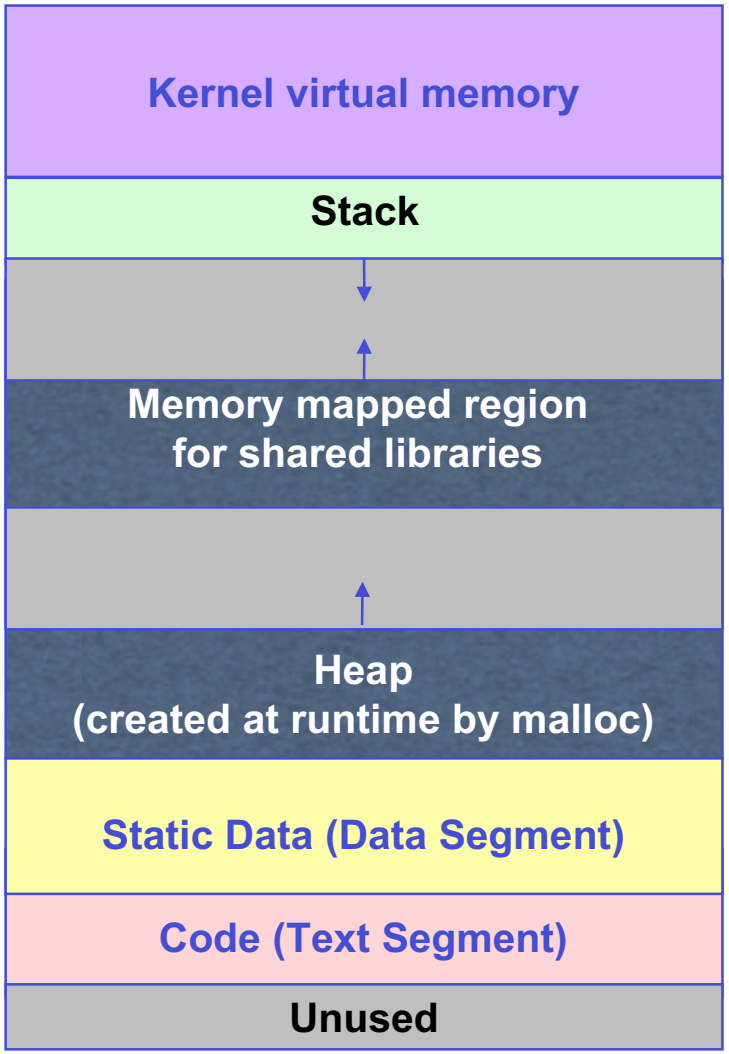
The Compiler Toolchain

We really on a sequence of tools to convert a high level language (like C) to machine code.



The Loader

- Once a program has been built into an executable, it needs to be loaded into memory to run.
- The loader's job is to allocate space for the program and to place it in memory.
- The code and data segments are defined in the executable.
- To learn more: CSC369 (operating systems)



The Linker

- We like to be able to write a program in multiple files.

- This lets us structure the code logically (in modules).
- It allows us to divide the work.
- It reduces compilation time, as only the files that are modified need to be re-compiled.

- The linker combines multiple files into a single executable.
  - This requires that it *patch symbol values.*

Assembling and Linking

```
$ cat other.c

$ cat main.c
#include <stdio.h>
int main(int argc, char* argv[]) {
    int x = ret_value(6, 3);
    printf("%d\n", x);
    return x;
}

$ gcc -c main.c

$ gcc -c other.c    # These made the compiler and assembler create object files -- .o files.
```

```
$ gcc other.o main.o  # This invokes the linker to combine the files.


$ ./a.out
6

$ # We explored adding prototypes (including erroneous ones) and found that the linker never complained!

$ # It's the compiler's job to find type errors. The linker does not check, so by default, the linker doesn't warn you if two functions don't match in type!
```

The Assembler

- The assembler converts assembly code into machine code.
- Assemblers are architecture specific.
  - There are MIPS assemblers, x86 assemblers, etc.

- Sometimes, the code being translated is not assembly -- it's some other intermediate form produced by the compiler.
  - i.e., LLVM's typed intermediate language

The Compiler

- The compiler's job is to convert high level code to assembly code.

- To learn more: CSC488 (compilers)

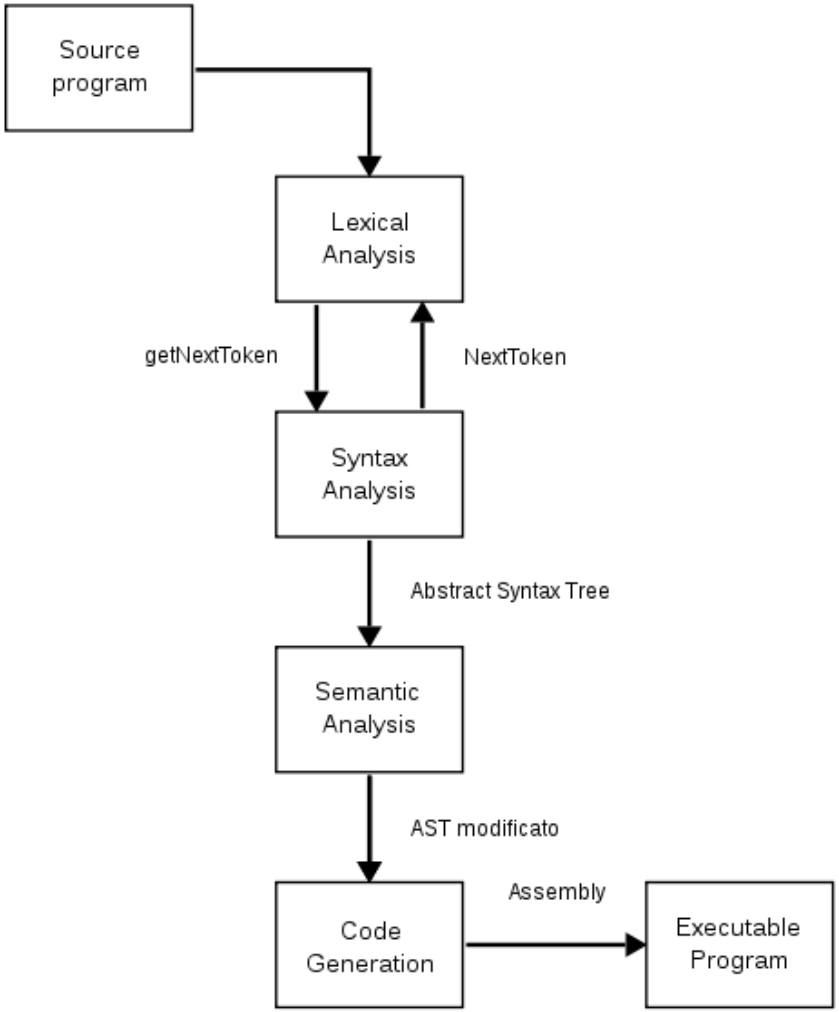


Image courtesy: http://commons.wikimedia.org/wiki/File:Multi-pass_compiler.svg

Compiling

```
int main(int argc, char* argv[]) {
    return 0;
}
```

```
$ gcc -S -march=mips32 main.c
$ # Don't try that at home. Or do, but expect
it that it might fail. I installed a version of gcc
to "cross-compile" to MIPS.
$ cat main.s
        .file   1 "main.c"
        .section .mdebug.abi32
        .previous
        .abicalls
        .text
        .align  2
        .globl  main
        .ent    main
main:
```

```
addiu   $sp,$sp,-8
sw      $fp,0($sp)
move    $fp,$sp
sw      $4,8($fp)
sw      $5,12($fp)
move    $2,$0
move    $sp,$fp
lw      $fp,0($sp)
addiu   $sp,$sp,8
j       $31
nop

.set    macro
.set    reorder
.end    main
.size   main, .-main
.ident  "GCC: (GNU) 4.1.2"
```

One Extra Stage: Preprocessing

```
#define RETVAL 3
int main(int argc, char* argv[]) {
    return RETVAL;
}

$ gcc -E main.c
# 1 "main.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "main.c"

int main(int argc, char* argv[]) {
    return 3;
}
```

Functions

Discussion: Reading "Quiz"

1. What are the names of the tools in the compiler toolchain? (And what is the rough responsibility of each?)

2. What does the *jal* (jump and link) instruction do?

3. List the expectations we, as programmers, have of functions. Here's one …
   a. After a function executes, control returns to the instruction *after* the function call at the call site.
   b. …

Key Questions

- What steps are required on the *caller*'s side to invoke a function?

- What steps are required on the *callee*'s side to execute and return from the function?

Functions

```
int my_func(int x, int y) {
    ...
    return z;
}
```

- Functions allow us to structure -- and reuse -- code
- Functions are governed by *conventions* that were affected by hardware limitations:
  Only a single value is returned, arguments may be passed, local variables do not escape the environment, ...
- They are also constrained by the compiler.
  - We need firm rules about how to transfer control to and from functions.

Implementing Functions

- What does the toolchain tell us about functions?
  - Functions may be defined in multiple files.
  - They may be invoked in files that do not contain the definitions.

- The location of the function will be patched during linking
- At the compilation stage, we need a description of the function and a standard way to call it.
  - We need *conventions*, and we need a *prototype*.

A Function in C

- In C, the declaration is the prototype.
  - It allows the invocation of the function without the definition.
- What do we expect to occur when the function is called?

Declaration:

```
int my_func(int, int);
```

Definition:

```
int my_func(int x, int y) {
    ...
    return z;
}
```

Use:

```
int my_var = my_func(3, 4);
```

Function Requirements

- Every function call needs its own distinct set of local variables, and these variables disappear when the function ends.
  - Variables are located in a *stack frame*.
- Functions must be invoked on arguments.
  - The argument values must be transferred.
- Every function call needs to return to the location that invoked it.
  - We must store a *return address* and *return value*.

Warning: Conventions Differ!

- The whole point of a *convention* is to have an agreement about the interface.

- … but there have been several versions of MIPS.
  - Different tools will target different versions.

- In particular, in the lab (next week), I am asking you to implement the version in A.6 of Larus's MIPS reference.
  - The gcc version we will explore today differs slightly.

Key Elements

- The frame pointer is used to point to the top of the stack frame. The stack pointer points to the bottom of the frame. (Why do we need both?)
- The caller must save some registers (the "caller saves"), and the callee must save other registers (the "callee saves"). The callee usually also stores arguments on the stack. Sometimes it also saves the $ra. When must it do so?
- The return value is placed in $v0.
- The callee then calls jr $ra to return back to the caller.

Key Elements: Caller

- Ensure that critical registers like $ra have been saved.
- Save “caller-save” registers.
- Place arguments into $a0-$a3 and on the stack.
- “Jump-and-link” to the function.

  ...
- On return, save the return value.
- Remove arguments from the stack.
- Restore caller-save registers.

Key Elements: Callee

- Allocate all of the space needed for the stack.
- If calling a function: store $ra and, if used, $fp.
  - It's good practice to save arguments, too.
- Move the $fp.
- Store callee-save registers.

  ...
- Place the return value in $v0.
- Restore callee-saves.
- Tear down the stack.

Caller- and Callee- Save Registers

- *Why should we separate the register file into two sets of registers?*

Caller- and Callee- Save Registers

- Why should we separate the register file into two sets of registers?
  - Performance!

- Caller-save registers are for very short-term temporaries.
  - The caller should not often need to save them.
- Callee-saves are for values that need to exist across function calls.
  - The callee only needs to save them if they are used.

Exceptions

Key Questions

- What is the purpose of an exception?
  - What is a system call?

- How are function calls and exceptions similar? How are they different?
  - How is an exception invoked? How do you return?

System Calls

- A system call is just a function call
  - ... that invokes the operating system.

- Printing, getting input, and allocating memory are all system call operations.
- We can't just give user programs control over thse operations because they must be shared across multiple users.

Invoking a System Call

- For security, the operating system must be able to access memory and execute instructions that the user cannot.
  - Otherwise, what would stop the user from just using a resource, rather than asking the OS to do so?

- A system call is invoked by placing arguments in the registers and then causing an *exception*.
  - The exception places the processor in a privileged mode -- one that can act as the OS.

Exceptions in MIPS

- An exception can be caused by a program or by an event.
  - For example, when you divide by zero, that causes an exception. syscall also creates an exception.
- When a syscall occurs, the cause of the exception is placed in a special register -- accessed using mfc0.
- Then, an exception handler is invoked.
  - System calls are sent directly to a syscall handler.

Exception Registers

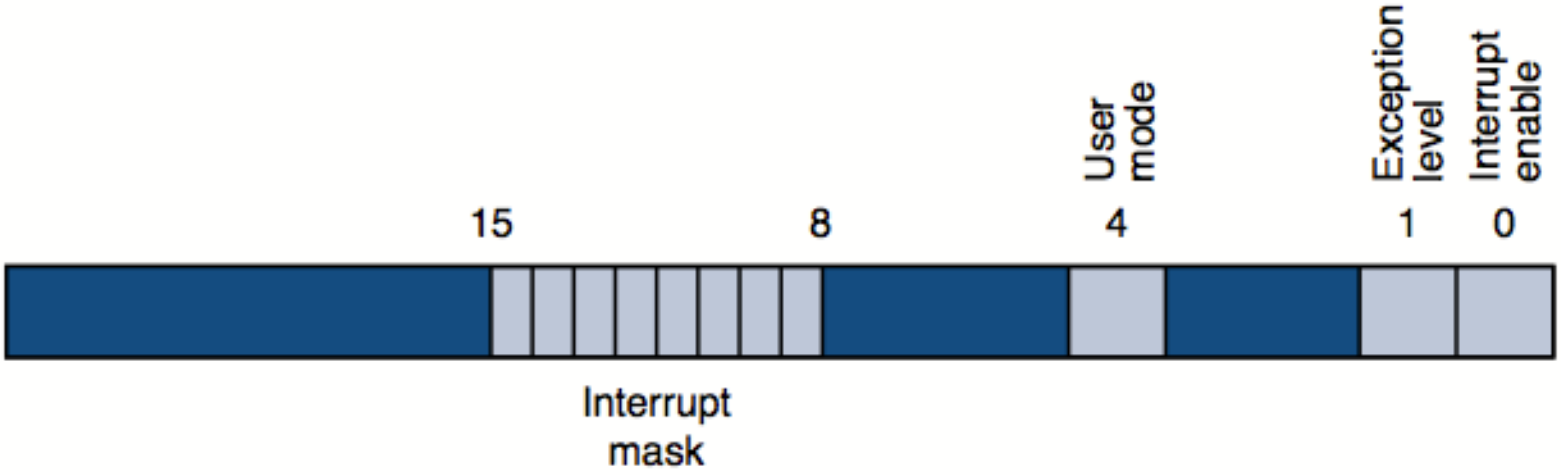


FIGURE A.7.1 The Status register.

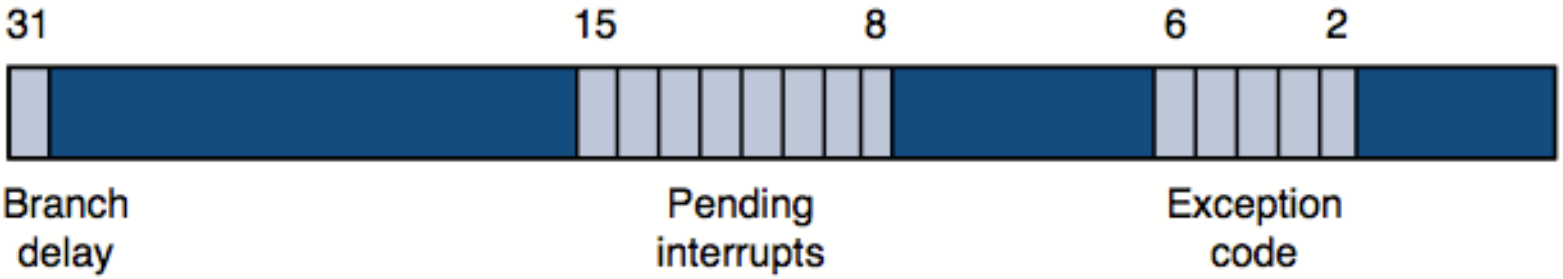


FIGURE A.7.2 The Cause register.

Source: Larus, *Assemblers, Linkers, and the SPIM Simulator*. http://pages.cs.wisc.edu/~larus/HP_AppA.pdf

Things to Remember

- The exception handler is just assembly code.
  - It's like any other function!
- **It must not cause an error.**

- It should leave the machine just as it was -- except for fixing whatever condition caused the exception.
- It can't change the stack. It shouldn't junk any registers.
  - The exception handler you'll use in the last lab saves two registers in the data segment. Those are the only registers you may use.