Introduction to C

CSE 410
Lecture 05
C Overview

- **C is a Higher Level Language (HLL)**
  - Assembler:
    - 1 assembler instruction → 1 machine instruction
    - Main job: turn a human readable instruction into

- **C is much more convenient** to write than assembler, but C’s semantics “expose” some aspects of the underlying hardware
  - In particular, main memory
C Overview

- The C compiler does the following things
  - **Generates instructions**
    - assembler/machine instructions that achieve the same computation as the C language program
  - **Allocates memory for variables**
  - **Manages which registers** are used to hold what values at each point in the assembly/machine code
  - *(other things we’re deferring to later discussion)*
#include <stdio.h>  // Somewhat like Java import  
#include <stdlib.h>  

int main(int argc, char *argv[]) {  // somewhat like public static void main(String[] args) 
    return EXIT_SUCCESS;  // EXIT_SUCCESS is a symbolic constant defined in stdlib.h 
}
Basic Statements are Like Java

- int var declaration: int x;
- assignment: x = y * 3 + z;
- for loop: for (i=0; i<10; i++) { ... }
- while loop: while (a[i] != 0) { ... }
- if-then-else: if (x<0) { y = -x; } else { y = x; }
Methods

- There are no classes
  - There are no objects

- All methods are public static

- All methods are in a single, flat, global namespace
  - You can’t have two methods named update()

- There is no polymorphism
  - You can’t have int update(int x) and also int update(int x, int y)
Variables

- There are procedure local variables, exactly like in Java
  ```c
  int invert(int x) {
    int y = -x;
    return y;
  }
  ```
- Because there are no objects, there are no instance variables shared across procedures
- But, there are global variables
- Global variables correspond to the .data segment from assembler
Global Variables

```c
int x = 0;
int y = 4;
int silly(int z) {
    x = y + z;
}
int sillier(int z) {
    x = y + 2*z;
}
```
How Does This Work

- At run time, memory is used like this:

```
local variables
unused
global variables
instructions
```

managed at runtime
.data segment
.text segment
Register Names

- There is a *convention* observed by all software regarding the use of registers
- It’s helpful for the registers’ symbolic names in assembly to reflect their conventional use

<table>
<thead>
<tr>
<th>Register</th>
<th>ABI Name</th>
<th>Description</th>
<th>Saver</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0</td>
<td>zero</td>
<td>Hard-wired zero</td>
<td>—</td>
</tr>
<tr>
<td>x1</td>
<td>ra</td>
<td>Return address</td>
<td>Caller</td>
</tr>
<tr>
<td>x2</td>
<td>sp</td>
<td>Stack pointer</td>
<td>Callee</td>
</tr>
<tr>
<td>x3</td>
<td>gp</td>
<td>Global pointer</td>
<td>—</td>
</tr>
<tr>
<td>x4</td>
<td>tp</td>
<td>Thread pointer</td>
<td>—</td>
</tr>
<tr>
<td>x5</td>
<td>t0</td>
<td>Temporary/alternate link register</td>
<td>Caller</td>
</tr>
<tr>
<td>x6-7</td>
<td>t1-2</td>
<td>Temporaries</td>
<td>Caller</td>
</tr>
<tr>
<td>x8</td>
<td>s0/fp</td>
<td>Saved register/frame pointer</td>
<td>Callee</td>
</tr>
<tr>
<td>x9</td>
<td>s1</td>
<td>Saved register</td>
<td>Callee</td>
</tr>
<tr>
<td>x10-11</td>
<td>a0-1</td>
<td>Function arguments/return values</td>
<td>Caller</td>
</tr>
<tr>
<td>x12-17</td>
<td>a2-7</td>
<td>Function arguments</td>
<td>Caller</td>
</tr>
<tr>
<td>x18-27</td>
<td>s2-11</td>
<td>Saved registers</td>
<td>Callee</td>
</tr>
<tr>
<td>x28-31</td>
<td>t3-6</td>
<td>Temporaries</td>
<td>Caller</td>
</tr>
</tbody>
</table>
Example C Programs

- A sequence of tiny examples we’ll compile and look at assembly code produced for them
Procedure Call

- **Caller calls callee**
  - `x = sub(y, z);
  int sub(int x, int y) { ... return z;};`

- **Calling a procedure involves a few steps**
  - Put the arguments somewhere the callee can find them
  - Set PC to the address of the procedure
  - Use the same CPU/registers as the caller to compute some result
  - Return the result to the caller
  - Go back to the caller at the instruction following the call
Put the Arguments Somewhere

- The caller puts the arguments in registers a0-a7
  - Why?
    - The called code will be using the same CPU and register set as the caller
    - Leaving arguments in registers (may) avoid caller having to store them in memory and then callee loading them back into registers
  - But... if callee itself calls some other method, it needs to use the same a0-a7 for its call
    - Callee may need to save parameter values in memory, so it can reuse a0-a7
  - Note: parameters are local variables
Set PC to Address of the Procedure / Go Back to the Caller

- There is a special instruction: jump-and-link
  jal ra, sub

- It does two things
  - jumps: sets the PC to the address of sub
  - links: sets register ra to value of the already updated PC
    - That is, saves the address of “the instruction following the call”

- The callee expects register RA to have the return address
  - When done, the caller jumps to return address from RAs
Allocate space for local variables

- The stack grows towards lower addresses
- By convention, register SP always points to the “top of the stack”
- To allocate space, callee just decrements SP
  
  ```
  addi sp, sp, -8
  ```
- To free space, it increments it
  
  ```
  addi sp, sp, 8
  ```
Use the same CPU/registers as the caller

- What register contents can the caller expect when callee returns to it?
  - (1) Could assume that callee has clobbered all the registers
    - caller would need to re-load values from memory
  - (2) Could require callee to return all registers unmodified
    - Callee would have to save all registers it uses on stack
  - (3) What’s done:
    - By convention, some registers are “callee saved” and some are “caller saved”
    - Registers s0-s11 are saved by caller
    - Space is allocated to save them on stack
    - They’re restored before returning
    - Don’t need to save if you don’t use them
Return the result to the caller

- By convention, callee uses register a0 to return result
Example: Calling a Function

void printInt(int);
int fib(int f1, int f2, int cnt);

int main(int argc, char *argv[]) {
    int result = fib( 1, 1, 10 );
    printInt(result);

    return 0;
}

int fib(int f1, int f2, int cnt) {
    if (cnt == 0) return f2;
    return fib(f2, f1+f2, cnt-1);
}

main:
addi sp,sp,-48
sw ra,44(sp)
sw s0,40(sp)
addi s0,sp,48
sw a0,-36(s0)
sw a1,-40(s0)
li a2,10
li a1,1
li a0,1
call fib
sw a0,-20(s0)
...
Example: Calling a Function

```c
int main(int argc, char *argv[]) {
    int result = fib(1, 1, 10);
    printInt(result);
    return 0;
}
```

```
li    a2,10
li    a1,1
li    a0,1
jal   ra, fib
sw    a0,-20(s0)
```
Example: Entering a Function

```c
int main(int argc, char *argv[]) {
    ...
}

int fib(int f1, int f2, int cnt) {
    if ( cnt == 0 ) return f2;
    return fib( f2, f1+f2, cnt-1 );
}
```

```
addi sp,sp,-32
sw ra,28(sp)
sw fp,24(sp)
addi fp,sp,32
sw a0,-20(s0)
sw a1,-24(s0)
sw a2,-28(s0)
```
Example: Entering a Function

```c
int main(int argc, char *argv[]) {
    ... ra
}
```

```c
int fib(int f1, int f2, int cnt) {
    if ( cnt == 0 ) return f2;
    return fib( f2, f1+f2, cnt-1 );
}
```

```
addi sp,sp,-32
sw ra,28(sp)
sw fp,24(sp)
addi fp,sp,32
sw a0,-20(s0)
sw a1,-24(s0)
sw a2,-28(s0)
```
Example: Entering a Function

```c
int main(int argc, char *argv[]) {
    ... 
}
```

```c
int fib(int f1, int f2, int cnt) {
    if ( cnt == 0 ) return f2;
    return fib( f2, f1+f2, cnt-1 );
}
```

```assembly
addi  sp,sp,-32
sw    ra,28(sp)
sw    fp,24(sp)
addi  fp,sp,32
sw    a0,-20(s0)
sw    a1,-24(s0)
sw    a2,-28(s0)
```
Example: Entering a Function

```c
int main(int argc, char *argv[]) {
    ...
}

int fib(int f1, int f2, int cnt) {
    if ( cnt == 0 ) return f2;
    return fib( f2, f1+f2, cnt-1 );
}
```

```
addi sp,sp,-32
sw ra,28(sp)
sw fp,24(sp)
addi fp,sp,32
sw a0,-20(s0)
sw a1,-24(s0)
sw a2,-28(s0)
```
Example: Entering a Function

```c
int main(int argc, char *argv[]) {
    ...
}

int fib(int f1, int f2, int cnt) {
    if (cnt == 0) return f2;
    return fib(f2, f1+f2, cnt-1);
}
```

```
addi sp,sp,-32
sw ra,28(sp)
sw fp,24(sp)
addi fp,sp,32
sw a0,-20(s0)
sw a1,-24(s0)
sw a2,-28(s0)
```
Example: Entering a Function

```c
int main(int argc, char *argv[]) {
    ...
}

int fib(int f1, int f2, int cnt) {
    if ( cnt == 0 ) return f2;
    return fib( f2, f1+f2, cnt-1 );
}
```

Stack

```
sp  ra
fp  a2  10
     a1  1
     a2  1
addi  sp,sp,-32
sw    ra,28(sp)
sw    fp,24(sp)
addi  fp,sp,32
sw    a0,-20(s0)
sw    a1,-24(s0)
sw    a2,-28(s0)
```
Example: Entering a Function

```c
int main(int argc, char *argv[]) {
    ...
}
```

```c
int fib(int f1, int f2, int cnt) {
    if (cnt == 0) return f2;
    return fib(f2, f1+f2, cnt-1);
}
```

```
addi sp,sp,-32
sw ra,28(sp)
sw fp,24(sp)
addi fp,sp,32
sw a0,-20(s0)
sw a1,-24(s0)
sw a2,-28(s0)
```
Stack Frames

```c
int main(int argc, char *argv[]) {
    int result = fib(1, 1, 10);
    printInt(result);
    return 0;
}
```

stack frame for fib(1, 1, 10) call
Stack Frames

int fib(int f1, int f2, int cnt) {
    if (cnt == 0) return f2;
    return fib(f2, f1+f2, cnt-1);
}

stack frame for fib(1, 1, 10) call

stack frame for fib(1, 2, 9) call
Returning From a Function

mv     a0, a5
lw     ra, 28(sp)
lw     fp, 24(sp)
addi   sp, sp, 32
jr      ra  # return to caller

Stack

mv     a0, a5
lw     ra, 28(sp)
lw     fp, 24(sp)
addi   sp, sp, 32
jr      ra  # return to caller

Saved registers:
- Saved ra
- Saved fp
- Cnt
- F2
- F1

Stack frame:
- F2
- F1
- Cnt
- Saved ra
- Saved fp
- Stack pointer (sp)
- Frame pointer (fp)
Linking

- The C compiler considers source code one file at a time.
- For software engineering reasons, the code for an application will be in many files.
  - Example: main.c and printReport.c
- This makes it impossible for the compiler to produce completely specified machine code, because it can’t know the offsets of symbols in memory.
  - Will main() be at offset 0 or printReport()?  
  - If both main.c and printReport.c have global variable declarations, what offsets are they at in the data segment?