CS61C - Machine Structures

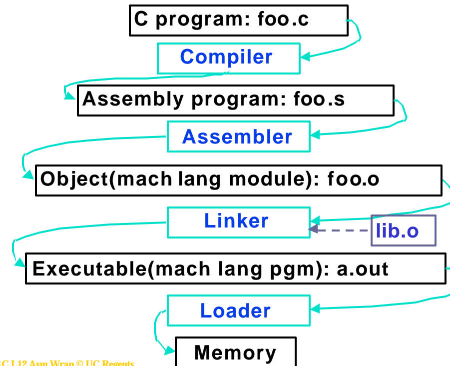
Lecture 12 - Assembly Wrapup, Pointers Revisited

October 6, 2000

David Patterson

http://www-inst.eecs.berkeley.edu/~cs61c/

Review (1/3)



Resolving References (1/2)

° Linker ***assumes*** first word of first text segment is at address 0x00000000.

° Linker knows:

- length of each text and data segment
- ordering of text and data segments

° Linker calculates:

- absolute address of each label to be jumped to (internal or external) and each piece of data being referenced

Resolving References (2/2)

° To resolve references:

- search for reference (data or label) in all symbol tables
- if not found, search library files (for example, for `printf`)
- once absolute address is determined, fill in the machine code appropriately

° Output of linker: executable file containing text and data (plus header)

Symbol Table Entries

° Symbol Table

Label	Address (of label)
`main:`	`0x00000000 (0x004001f0 later)`
`loop:`	`0x00000018`
`str:`	`0x0 (0x10004000 later)`
`printf:`	`0x0 (0x00400260 later)`

° Relocation Information (for external addr)

Instr. Address	Instr. Type	Symbol
`0x00000040`	`HI16`	`str`
`0x00000044`	`LO16`	`str`
`0x0000004c`	`jal`	`printf`

Example: C Þ Asm Þ Obj Þ Exe Þ Run

•Remove pseudoinstructions, assign addresses

```
00 addiu $29,$29,-32      30 addiu $8,$14, 1
04 sw    $31,20($29)      34 sw    $8,28($29)
08 sw    $4, 32($29)      38 slti  $1,$8, 101
0c sw    $5, 36($29)      3c bne   $1,$0, loop
10 sw    $0, 24($29)      40 lui   $4, hi.str
14 sw    $0, 28($29)      44 ori   $4,$4,lo.str
18 lw    $14, 28($29)     48 lw    $5,24($29)
1c multu $14, $14         4c jal   printf
20 mflo  $15              50 add   $2, $0, $0
24 lw    $24, 24($29)     54 lw    $31,20($29)
28 addu  $25,$24,$15      58 addiu $29,$29,32
2c sw    $25, 24($29)     5c jr    $31
```

Outline

- Signed vs. Unsigned MIPS Instructions
- Pseudoinstructions
- C case statement and MIPS code
- Multiply/Divide
- Problems with Pointers
- Multiply/Divide
- Faculty debate on pointers (if time permits)

Loading, Storing bytes

- In addition to word data transfers (`lw`, `sw`), MIPS has byte data transfers:
- load byte: `lb`
- store byte: `sb`
- same format as `lw`, `sw`

Loading, Storing bytes

- What do with other 24 bits in the 32 bit register?
  - `lb`: sign extends to fill upper 24 bits
- Suppose byte at 100 has value 0x0F, byte at 200 has value 0xFF

```
lb $s0, $zero(100) ; $s0 = ??
lb $s1, $zero(200) ; $s1 = ??
```

- Multiple choice: $s0? $s1?

a) 15; b) 255; c) -1; d) -255; e) -15

Loading bytes

- Normally with characters don't want to sign extend
- So MIPS instruction that doesn't sign extend when loading bytes:
- load byte unsigned: `lbu`

Overflow in Arithmetic (1/2)

- Reminder: Overflow occurs when there is a mistake in arithmetic due to the limited precision in computers.
- Example (4-bit unsigned numbers):

+15	1111
+3	0011
+18	10010

  - But we don't have room for 5-bit solution, so the solution would be 0010, which is +2, which is wrong.

Overflow in Arithmetic (2/2)

- Some languages detect overflow (Ada), some don't (C)
- MIPS solution is 2 kinds of arithmetic instructions to recognize 2 choices:
  - add (`add`), add immediate (`addi`) and subtract (`sub`) cause overflow to be detected
  - add unsigned (`addu`), add immediate unsigned (`addiu`) and subtract unsigned (`subu`) do not cause overflow detection
- Compiler selects appropriate arithmetic
  - MIPS C compilers produce `addu`, `addiu`, `subu`

Unsigned Inequalities

° Just as unsigned arithmetic instructions:

`addu, subu, addiu`

(really "don't overflow" instructions)

° There are unsigned inequality instructions:

`sltu, sltiu`

but really do mean unsigned compare!

° 0x80000000 < 0x7fffffff signed (slt, slti)

° 0x80000000 > 0x7fffffff unsigned (sltu, sltiu)

Number Representation for I-format

op	rs	rt	address/immediate
6 bits	5 bits	5 bits	16 bits

° Loads, stores treat the address (0x8000 to 0x7FFF) as a 16-bit 2's complement number: -2^{15} to $2^{15}-1$ or -32768 to +32767 added to $rs

- Hence $gp set to 0x10001000 so can easily address from 0x10000000 to 0x10001111

° Most immediates represent same values: addi, addiu, slti, sltiu (including "unsigned" instrs addiu,sltiu!)

° andi, ori consider immediate a 16-bit unsigned number: 0 to $2^{16}-1$, or 0 to 65535 (0x0000 to 0x1111)

True Assembly Language (1/3)

° **Pseudoinstruction**: A MIPS instruction that doesn't turn directly into a machine language instruction.

° What happens with pseudoinstructions?

- They're broken up by the assembler into several "real" MIPS instructions.
- But what is a "real" MIPS instruction?

True Assembly Language (2/3)

° **MAL** (MIPS Assembly Language): the set of instructions that a programmer may use to code in MIPS; this *includes* pseudoinstructions

° **TAL** (True Assembly Language): the set of instructions that can actually get translated into a single machine language instruction (32-bit binary string)

° A program must be converted from MAL into TAL before it can be translated into 1s and 0s.

True Assembly Language (3/3)

° Problem:

- When breaking up a pseudoinstruction, the assembler will usually need to use an extra register.
- If it uses any regular register, it'll overwrite whatever the program has put into it.

° Solution:

- Reserve a register ($1 or $at) that the assembler will use when breaking up pseudoinstructions.
- Since the assembler may use this at any time, it's not safe to code with it.

The C Switch Statement (1/3)

° Choose among four alternatives depending on whether k has the value 0, 1, 2 or 3. Compile this C code:

```
switch (k) {
   case 0: f=i+j; break; /* k=0*/
   case 1: f=g+h; break; /* k=1*/
   case 2: f=g–h; break; /* k=2*/
   case 3: f=i–j; break; /* k=3*/
   }
```

Example: The C Switch Statement (2/3)

- This is complicated, so simplify.
- Rewrite it as a chain of if-else statements, which we already know how to compile:

```
if(k==0) f=i+j;
  else if(k==1) f=g+h;
    else if(k==2) f=g–h;
      else if(k==3) f=i–j;
```

- Use this mapping:

```
f: $s0, g: $s1, h: $s2, i: $s3,
 j: $s4, k: $s5
```


Example: The C Switch Statement (3/3)

- Final compiled MIPS code:

```
     bne  $s5,$0,L1     # branch k!=0
     add  $s0,$s3,$s4   #k==0 so f=i+j
     j    Exit # end of case so Exit
L1:  addi $t0,$s5,-1    # $t0=k-1
     bne  $t0,$0,L2     # branch k!=1
     add  $s0,$s1,$s2   #k==1 so f=g+h
     j    Exit # end of case so Exit
L2:  addi $t0,$s5,-2    # $t0=k-2
     bne  $t0,$0,L3     # branch k!=2
     sub  $s0,$s1,$s2   #k==2 so f=g-h
     j    Exit # end of case so Exit
L3:  addi $t0,$s5,-3    # $t0=k-3
     bne  $t0,$0,Exit   # branch k!=3
     sub  $s0,$s3,$s4   #k==3 so f=i-j
Exit:
```


Common Problems with Pointers: Hilfinger

- 1. Some people do not understand the distinction between `x = y` and `*x = *y`
- 2. Some simply haven't enough practice in routine pointer-hacking, such as how to splice an element into a list.
- 3. Some do not understand the distinction between `struct Foo x;` and `struct Foo *x;`
- 4. Some do not understand the effects of `p = &x` and subsequent results of assigning through dereferences of `p`, or of deallocation of `x`.

Address vs. Value

- Fundamental concept of Comp. Sci.
- Even in Spreadsheets: select cell A1 for use in cell B1

	A	B
1	100	100
2		

- Do you want to put the address of cell A1 in formula (=A1) or A1's value (100)?
- Difference? When change A1, cell using address changes, but not cell with old value

Address vs. Value in C

- Pointer: a variable that contains the address of another variable
  - HLL version of machine language address
- Why use Pointers?
  - Sometimes only way to express computation
  - Often more compact and efficient code
- Why not? (according to Eric Brewer)
  - Huge source of bugs in real software, perhaps the largest single source
  - 1) Dangling reference (premature free)
  - 2) Memory leaks (tardy free): can't have long-running jobs without periodic restart of them

C Pointer Operators

- Suppose `c` has value 100, located in memory at address 0x10000000
- Unary operator `&` gives address: `p = &c;` gives address of `c` to `p`;
  - `p` "points to" `c`
  - `p == 0x10000000`
- Unary operator `*` gives value that pointer points to: if `p = &c;` then
  - "Dereferencing a pointer"
  - `* p == 100`

Assembly Code to Implement Pointers

° deferencing ⇒ data transfer in asm.

- ... = ... *p ...; ⇒ load
  (get value from location pointed to by p)
  load word (lw) if int pointer,
  load byte unsigned (lbu) if char pointer
- *p = ...; ⇒ store
  (put value into location pointed to by p)

Assembly Code to Implement Pointers

° c is int, has value 100, in memory at address 0x10000000, p in $a0, x in $s0

```
 p = &c;   /* p gets 0x10000000 */
 x = *p;   /* x gets 100 */
*p = 200;  /* c gets 200 */
```

```
 # p = &c;   /* p gets 0x10000000 */
lui $a0,0x1000 # p = 0x10000000

 # x = *p;   /* x gets 100 */
lw  $s0, 0($a0) # dereferencing p

# *p = 200; /* c gets 200 */
addi $t0,$0,200
sw   $t0, 0($a0) # dereferencing p
```


Registers and Pointers

° Registers do not have addresses

⇒ registers cannot be pointed to

⇒ cannot allocate a variable to a register if it may have a pointer to it

C Pointer Declarations

° C requires pointers be declared to point to particular kind of object (int, char, ...)

° Why? Safety: fewer problems if cannot point everywhere

° Also, need to know size to determine appropriate data transfer instruction

° Also, enables pointer calculations

- Easy access to next object: p+1
- Or to i-th object: p+i
- Byte address? multiplies i by size of object

C vs. Asm

```
int strlen(char *s) {
 char *p = s; /* p points to chars */

  while (*p != '\0')
   p++; /* points to next char */
 return p - s; /* end - start */
}
```

```
      mov  $t0,$a0
      lbu $t1,0($t0) /* derefence p */
      beq  $t1,$zero, Exit

Loop: addi $t0,$t0,1  /* p++ */
      lbu $t1,0($t0) /* derefence p */
      bne  $t1,$zero, Loop

Exit: sub $v0,$t1,$a0
      jr $ra
```


C pointer "arithmetic"

° What arithmetic OK for pointers?

- Add an integer to a pointer: p+i
- Subtract 2 pointers (in same array): p-s
- Comparing pointers (<, <=, ==, !=, >, >=)
- Comparing pointer to 0: p == 0
  (0 used to indicate it points to nothing; used for end of linked list)

° Everything else illegal (adding 2 pointers, multiplying 2 pointers, add float to pointer, ...)

- Why? Makes no sense in a program

Common Pointer Use

° Array size n; want to access from 0 to n-1, but test for exit by comparing to address one element past the array

```
int a[10], *q, sum = 0;

...
p = &a[0]; q = &a[10];
while (p != q)
   sum = sum + *p++;
```

- Is this legal?

° C defines that one element past end of array must be a valid address, i.e., not cause an bus error or address error

Common Pointer Mistakes

° Common error; Declare and write:

```
int *p;
*p = 10; /* WRONG */
```

- What address is in p? (NULL)

° C defines that memory location 0 must not be a valid address for pointers

- NULL defined as 0 in `<stdio.h>`

Common Pointer Mistakes

° Copy pointers vs. values:

```
int *ip, *iq, a = 100, b = 200;
ip = &a; iq = &b;
*ip = *iq; /* what changed? */
ip = iq;   /* what changed? */
```


Pointers and Heap Allocated Storage

° Need pointers to point to `malloc()` created storage

° What if free storage and still have pointer to storage?

- "Dangling reference problem"

° What if don't free storage?

- "Memory leak problem"

Multiple pointers to same object

° Multiple pointers to same object can lead to mysterious behavior

° `int *x, *y, a = 10, b;`

```
...
y = & a;
...
x = y;
...
*x = 30;
...
/* no use of *y */
printf("%d", *y);
```


Java doesn't have these pointer problems?

° Java has automatic garbage collection, so only when last pointer to object disappears, object is freed

° Point 4 above not a problem in Java:

> "4. Some do not understand the effects of p = &x and subsequent results of assigning through dereferences of p, or of deallocation of x."

° What about 1, 2, 3, according to Hilfinger?

Multiplication (1/3)

° Paper and pencil example (unsigned):

```
Multiplicand  1000
Multiplier   x1001
              1000
             0000
            0000
          +1000
          01001000
```

• m bits x n bits = m + n bit product

Multiplication (2/3)

° In MIPS, we multiply registers, so:

• 32-bit value x 32-bit value = 64-bit value

° Syntax of Multiplication:

• `mult` register1, register2

• Multiplies 32-bit values in specified registers and puts 64-bit product in special result registers:

- puts upper half of product in hi
- puts lower half of product in lo

• hi and lo are 2 registers separate from the 32 general purpose registers

Multiplication (3/3)

° Example:

• in C: `a = b * c;`

• in MIPS:

- let b be $s2; let c be $s3; and let a be $s0 and $s1 (since it may be up to 64 bits)

```
mult $s2,$s3   # b*c
mfhi $s0       # upper half of
  # product into $s0
mflo $s1       # lower half of
  # product into $s1
```

° Note: Often, we only care about the lower half of the product.

Division (1/3)

° Paper and pencil example (unsigned):

```
                 1001     Quotient
Divisor 1000|1001010     Dividend
            -1000
                10
                101
                1010
               -1000
                  10     Remainder
             (or Modulo result)
```

° Dividend = Quotient x Divisor + Remainder

Division (2/3)

° Syntax of Division:

• `div` register1, register2

• Divides 32-bit values in register 1 by 32-bit value in register 2:

- puts remainder of division in hi
- puts quotient of division in lo

° Notice that this can be used to implement both the C division operator (/) and the C modulo operator (%)

Division (3/3)

° Example:

• in C: `a = c / d;`
`b = c % d;`

• in MIPS:

- let a be $s0; let b be $s1; let c be $s2; and let d be $s3

```
div  $s2,$s3   # lo=c/d, hi=c%d
mflo $s0       # get quotient
mfhi $s1       # get remainder
```

More Overflow Instructions

- In addition, MIPS has versions for these two arithmetic instructions which do not detect overflow:

```
multu
divu
```

- Also produces unsigned product, quotient, remainder

Common Problems with Pointers: Brewer

- 1) heap-based memory allocation (malloc/free in C or new/delete in C++) is a huge source of bugs in real software, perhaps the largest single source. The worse problem is the dangling reference (premature free), but the lesser one, memory leaks, mean that you can't have long-running jobs without restarting them periodically

Common Problems with Pointers: Brewer

- 2) aliasing: two pointers may have different names but point to the same value. This is really the more fundamental problem of pass by reference (i.e., pointers): people normally assume that they are the only one modifying the an object
- This is often not true for pointers -- there may be other pointers and thus other modifiers to your object. Aliasing is the special case where you have both of the pointers...

Common Problems with Pointers: Brewer

- In general, pointers tend to make it unclear if you are sharing an object or not, and whether you can modify it or not. If I pass you a copy, then it is yours alone and you can modify it if you like. The ambiguity of a reference is bad; particularly for return values such as getting an element from a set -- is the element a copy or the master version, or equivalently do all callers get the same pointer for the same element or do they get a copy. If it is a copy, where did the storage come from and who should deallocate it?

Java vs. C. vs. C++ Semantics?

1. The semantics of pointers in Java, C, and C++ are IDENTICAL. The difference is in what Java lacks: it does not allow pointers to variables, fields, or array elements. Since Java has no pointers to array elements, it has no "pointer arithmetic" in the C sense (a bad term, really, since it only means the ability to refer to neighboring array locations).

- When considering what Java, C, and C++ have in common, the semantics are the same.

Java pointer is different from meaning in C?

- 2. The meaning of "pointer semantics" is that after

```
y.foo = 3;
x = y;
x.foo += 1;
```

- y.foo is 4, whereas after

```
x = z;
x.foo += 1;
```

- y and y.foo are unaffected.
- This is true in Java, as it is true in C/C++ (except for their use of -> instead of '.').

Java vs. C pass by reference?

- 3. NOTHING is passed by reference in Java, just as nothing is passed by reference in C. A parameter is "passed by reference" when assignments to a parameter within the body means assignment to the actual value. In Java and legal C, for a local variable `x` (whose address is never taken) the initial and final values of x before and after

  …`x;` … `f(x);` … `x;` …

- are identical, which is the definition of "pass by value".

What about Arrays, Prof. Hilfinger?

- 3. There is a common misstatement that "arrays are passed by reference in C". The language specification is quite clear, however: arrays are not passed in C at all.
  - (If you want to "pass an array" you must pass a pointer to the array, since you can't pass an array at all)
- The semantics are really very clean---ALL values, whether primitive or reference---obey EXACTLY the same rule. We really HAVE to refrain from saying that "objects are passed by reference", since students have a hard enough time understanding that `f(x)` can't reassign `x` as it is.

"And in Conclusion.." 1/2

- Pointer is high level language version of address
  - Powerful, dangerous concept
- Like goto, with self-imposed discipline can achieve clarity and simplicity
  - Also can cause difficult to fix bugs
- C supports pointers, pointer arithmetic
- Java structure pointers have many of the same potential problems!

"And in Conclusion.." 2/2

- MIPS Signed v. Unsigned "overloaded" term
  - Do/Don't sign extend (lb, lbu)
  - Don't overflow (addu, addiu, subu, multu, divu)
  - Do signed/unsigned compare (slt,slti/sltu,sltiu)
  - Immediate sign extension independent of term (andi, ori zero extend; rest sign extend)
- Assembler uses $at to turn MAL into TAL
- Compiler transitions between levels of abstraction
- Next: Input/Output (COD chapter 8)