inst.eecs.berkeley.edu/~cs61c

CS61C : Machine Structures

Lecture #14 – CALL II, 1st Half review

2008-7-15

Script browsers in C?

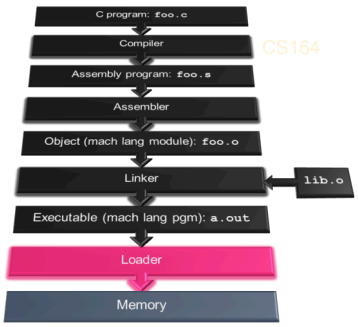
http://tech.slashdot.org/article.pl?sid=08/07/07/1724236

Albert Chae, Instructor

Review

- Interpretation – less efficient, easier to program, debug
- Translation – more efficient, harder to read/program, can protect source
- Translating C Programs
  - Compiler
  - Assembler
  - Linker
  - Loader

Where Are We Now?



Loader (1/3)

- Input: Executable Code (e.g., `a.out` for MIPS)
- Output: (program is run)
- Executable files are stored on disk.
- When one is run, loader's job is to load it into memory and start it running.
- In reality, loader is the operating system (OS)
  - loading is one of the OS tasks

Loader (2/3)

- So what does a loader do?
- Reads executable file's header to determine size of text and data segments
- Creates new address space for program large enough to hold text and data segments, along with a stack segment
- Copies instructions and data from executable file into the new address space (this may be anywhere in memory as we'll see later)

Loader (3/3)

- Copies arguments passed to the program onto the stack
- Initializes machine registers
  - Most registers cleared, but stack pointer assigned address of 1st free stack location
- Jumps to start-up routine that copies program's arguments from stack to registers and sets the PC
  - If main routine returns, start-up routine terminates program with the exit system call

Example: C⇒Asm⇒Obj⇒Exe ⇒Run

C Program Source Code: prog.c

```
#include <stdio.h>

int main (int argc, char *argv[]) {

int i, sum = 0;

 for (i = 0; i<= 100; i++)
    sum = sum + i * i;

printf ("The sum of sq from 0 .. 100
  is %d\n", sum);

}
```

"printf" lives in "libc"

Compilation: MAL

```
.text
  .align 2
  .globl main
main:
subu $sp,$sp,32
sw   $ra, 20($sp)
sd   $a0, 32($sp)
sw   $0, 24($sp)
sw   $0, 28($sp)
loop:
lw   $t6, 28($sp)
mul$t7, $t6,$t6
lw   $t8, 24($sp)
addu $t9,$t8,$t7
sw   $t9, 24($sp)
addu $t0, $t6, 1
sw   $t0, 28($sp)
ble $t0,100, loop
  la $a0, str
lw   $a1, 24($sp)
jalprintf
  move $v0, $0
lw   $ra, 20($sp)
addiu $sp,$sp,32
jr $ra
  .data
  .align 0
str:
  .asciiz"The sum
  of sq from 0 ..
  100 is %d\n"
```

Where are 7 pseudo-instructions?

Compilation: MAL

```
.text
  .align 2
  .globl main
main:
subu $sp,$sp,32
sw   $ra, 20($sp)
sd   $a0, 32($sp)
sw   $0, 24($sp)
sw   $0, 28($sp)
loop:
lw   $t6, 28($sp)
mul$t7, $t6,$t6
lw   $t8, 24($sp)
addu $t9,$t8,$t7
sw   $t9, 24($sp)
addu $t0, $t6, 1
sw   $t0, 28($sp)
ble $t0,100, loop
la   $a0, str
lw   $a1, 24($sp)
jalprintf
move $v0, $0
lw   $ra, 20($sp)
addiu $sp,$sp,32
jr $ra
  .data
  .align 0
str:
.asciiz  "The sum
  of sq from 0 ..
  100 is %d\n"
```

7 pseudo-instructions underlined

Assembly step 1:

Remove pseudoinstructions, assign addresses

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

Assembly step 2

Create relocation table and symbol table

- **Symbol Table**

Label	address (in module)	type
main:	0x00000000	global text
loop:	0x00000018	local text
str:	0x00000060	local data

- **Relocation Information**

Address	Instr. type	Dependency
0x00000040	lui	l.str
0x00000044	ori	r.str
0x0000004c	jal	printf

Assembly step 3

Resolve local PC-relative labels

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

Assembly step 4

- Generate object (`.o`) file:
  - Output binary representation for
    - ext segment (instructions),
    - data segment (data),
    - symbol and relocation tables.
  - Using dummy "placeholders" for unresolved absolute and external references.

Text segment in object file

```
0x000000   00100111101111011111111111100000
0x000004   10101111101111110000000000010100
0x000008   10101111101001000000000000100000
0x00000c   10101111101001010000000000100100
0x000010   10101111101000000000000000011000
0x000014   10101111101000000000000000011100
0x000018   10001111101011100000000000011100
0x00001c   10001111101111000000000000011000
0x000020   00000001110011100000000000011001
0x000024   00100101110010000000000000000001
0x000028   00101001000000010000000001100101
0x00002c   10101111101010000000000000011100
0x000030   00000000000000000111100000010010
0x000034   00000011000011111100100000100001
0x000038   00010100001000001111111111110111
0x00003c   10101111101110010000000000011000
0x000040   00111100000000100000000000000000
0x000044   10001111101001010000000000000000
0x000048   00001100000100000000000011101100
0x00004c   00100100000000000000000000000000
0x000050   10001111101111110000000000010100
0x000054   00100111101111010000000000100000
0x000058   00000011111000000000000000001000
0x00005c   00000000000000000001000000100001
```

Link step 1: combine `prog.o`, `libc.o`

- Merge text/data segments
- Create absolute memory addresses
- Modify & merge symbol and relocation tables
- Symbol Table

Label	Address
main:	0x00000000
loop:	0x00000018
str:	0x10000430
printf:	0x000003b0 …

- Relocation Information

Address	Instr. Type	Dependency
0x00000040	luil	str
0x00000044	orir	str
0x0000004c	jal	printf …

Link step 2:

- Edit Addresses in relocation table
  - *(shown in TAL for clarity, but done in binary)*

```
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, -10
10 sw    $0, 24($29)        40 lui   $4, 4096
14 sw    $0, 28($29)        44 ori   $4,$4,1072
18 lw    $14, 28($29)       48 lw    $5,24($29)
1c multu $14, $14           4c jal812
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
```

Link step 3:

- Output executable of merged modules.
  - Single text (instruction) segment
  - Single data segment
  - Header detailing size of each segment
- NOTE:
  - The preceeding example was a much simplified version of how ELF and other standard formats work, meant only to demonstrate the basic principles.

Peer Instruction

Which of the following instr. may need to be edited during link phase?

```
Loop: lui $at, 0xABCD      }# A
      ori $a0,$at, 0xFEDC  }
      jal add_link         # B
      bne $a0,$v0, Loop    # C
```

	ABC
1:	FFF
2:	FFT
3:	FTF
4:	FTT
5:	TFF
6:	TFT
7:	TTF
8:	TTT

Peer Instruction Answer

Which of the following instr. may need to be edited during link phase?

```
Loop: lui $at, 0xABCD      }# A
      ori $a0,$at, 0xFEDC  }
      jal add_link         # B
      bne $a0,$v0, Loop    # C
```

$a0 just holds a number; OK

subroutine; relocate

PC-relative branch; OK

	ABC
1:	FFF
2:	FFT
3:	FTF
4:	FTT
5:	TFF
6:	TFT
7:	TTF
8:	TTT

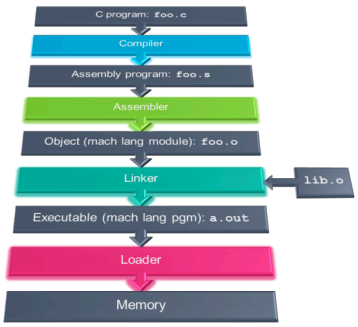
Administrivia

- Assignments
  - HW3 due 7/16 @ 11:59pm
  - Proj2 due 7/18 @ 11:59pm
- Grades
  - HW1,2, labs1-6 are up.
    - If not, contact reader (HWs) or TA (labs) right away.
    - We will have a grade freeze for these grades TBA.
  - HW0, quizzes1-6 will be posted by Thursday

Administrivia…Midterm

- Midterm Mon 2008-07-21@7-10pm, 155 Dwinelle
  - Bring pencils and eraser!
  - You can bring green sheet and one handwritten double sided note sheet
  - No calculator, laptop, etc.
- faux midterm: 7/16 @ 6-9pm 10 Evans
- review session: 7/17 in lecture (will go over faux exam)

Things to Remember (1/3)



Things to Remember (2/3)

- Compiler converts a single HLL file into a single assembly language file.
- Assembler removes pseudoinstructions, converts what it can to machine language, and creates a checklist for the linker (relocation table). This changes each .s file into a .o file.
  - Does 2 passes to resolve addresses, handling internal forward references
- Linker combines several .o files and resolves absolute addresses.
  - Enables separate compilation, libraries that need not be compiled, and resolves remaining addresses
- Loader loads executable into memory and begins execution.

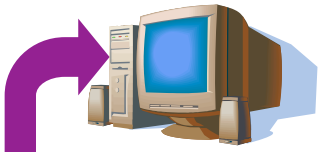
Things to Remember 3/3

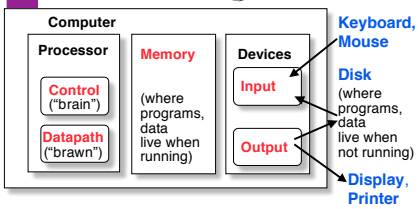
- Stored Program concept mean instructions just like data, so can take data from storage, and keep transforming it until load registers and jump to routine to begin execution
  - Compiler ⇒ Assembler ⇒ Linker (⇒ Loader)

1st Half Review

This is your chance to ask questions about anything from the past 4 weeks

Anatomy: 5 components of any Computer





Numbers: positional notation

- A digit's position determines how much value it adds to the whole number.
- Number Base B ⇒ B symbols per digit:
  - Base 10 (Decimal): 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
    Base 2 (Binary): 0, 1
- Number representation:
  - $d_{31}d_{30}$ ... d_1d_0 is a 32 digit number
  - value = $d_{31} \times B^{31} + d_{30} \times B^{30} + ... + d_1 \times B^1 + d_0 \times B^0$
- Binary: 0,1 (In binary, digits called "bits")
  - 0b11010 = $1\times2^4 + 1\times2^3 + 0\times2^2 + 1\times2^1 + 0\times2^0 = 26$
  - Here 5 digit binary # turns into a 2 digit decimal #
  - Can we find a base that converts to binary easily?

Hexadecimal Numbers: Base 16

- Hexadecimal:
  0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F
  - Normal digits + 6 more from the alphabet
  - In C, written as 0x… (e.g., 0xFAB5)
- Conversion: Binary⇔Hex
  - 1 hex digit represents 16 decimal values
  - 4 binary digits represent 16 decimal values
  - Þ1 hex digit replaces 4 binary digits
- One hex digit is a "nibble". Two is a "byte"

Decimal vs. Hexadecimal vs. Binary

Examples:

1010 1100 0011 (binary)
= 0xAC3

10111 (binary)
= 0001 0111 (binary)
= 0x17

0x3F9
= 11 1111 1001 (binary)

How do we convert between hex and Decimal?

00	0	0000
01	1	0001
02	2	0010
03	3	0011
04	4	0100
05	5	0101
06	6	0110
07	7	0111
08	8	1000
09	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

MEMORIZE!

kibi, mebi, gibi, tebi, pebi, exbi, zebi, yobi

`en.wikipedia.org/wiki/Binary_prefix`

- New IEC Standard Prefixes [only to exbi officially]

Name	Abbr	Factor
kibi	Ki	2^{10} = 1,024
mebi	Mi	2^{20} = 1,048,576
gibi	Gi	2^{30} = 1,073,741,824
tebi	Ti	2^{40} = 1,099,511,627,776
pebi	Pi	2^{50} = 1,125,899,906,842,624
exbi	Ei	2^{60} = 1,152,921,504,606,846,976
zebi	Zi	2^{70} = 1,180,591,620,717,411,303,424
yobi	Yi	2^{80} = 1,208,925,819,614,629,174,706,176

As of this writing, this proposal has yet to gain widespread use…

MEMORIZE!

The way to remember #s

- What is 2^{34}? How many bits addresses (i.e., what's `ceil` $\log_2$ = `lg` of) 2.5 TiB?
- Answer! 2^{XY} means…

X=0 ⇒ ---	Y=0 ⇒ 1
X=1 ⇒ kibi ~10^3	Y=1 ⇒ 2
X=2 ⇒ mebi ~10^6	Y=2 ⇒ 4
X=3 ⇒ gibi ~10^9	Y=3 ⇒ 8
X=4 ⇒ tebi ~10^{12}	Y=4 ⇒ 16
X=5 ⇒ pebi ~10^{15}	Y=5 ⇒ 32
X=6 ⇒ exbi ~10^{18}	Y=6 ⇒ 64
X=7 ⇒ zebi ~10^{21}	Y=7 ⇒ 128
X=8 ⇒ vobi ~10^{24}	Y=8 ⇒ 256
	Y=9 ⇒ 512

MEMORIZE!

What to do with representations of numbers?

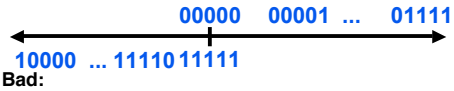
- Just what we do with numbers!
  - Arithmetic, comparisons
- Use them to represent ANYTHING
  - Characters – ASCII, UNICODE
  - Boolean – True/False
  - Colors, memory addresses, MIPS instructions
- With N bits ⇔ represent/index at most 2^N things
- With Y things to represent ⇔ need at least ceil(lg(Y)) bits to represent

How to Represent Negative Numbers?

- Define leftmost bit to be sign!
  - 0 ⇒ +, 1 ⇒ –
- sign and magnitude – left most bit is sign, rest of number is unsigned value
- Bad:
  - Arithmetic circuit complicated because adding 1 doesn't always result in bigger number
  - Also, two zeros
    - 0x00000000 = $+0_{ten}$
    - 0x80000000 = -0_{ten}

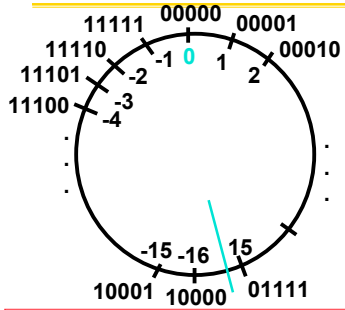
Another try: complement the bits

- Example: $7_{10} = 00111_2$ $-7_{10} = 11000_2$
- Called One's Complement
- Note: positive numbers have leading 0s, negative numbers have leadings 1s.
  - So leftmost bit is still a sign bit

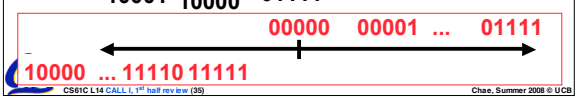


- Bad:
  - there are still 2 zeros
    - 0x00000000 = $+0_{ten}$
    - 0xFFFFFFFF = -0_{ten}
  - HW still more complicated than it needs to be

2's Complement Number "wheel": N = 5



- 2^{N-1} non-negatives
- 2^{N-1} negatives
- one zero



Standard Negative Number Representation

- Two's Complement – Shift one's complement over one to get rid of 2 zeros and make hardware simpler
- Features of two's complement:
  - To negate: invert all bits and add 1
  - Sign extension:
    - Easy to convert n bit number to m bit number, when m > n
    - Just copy sign bit over to the left
- If not enough bits to hold number, we get overflow

C Syntax: Variables

- Declare variables with type and name

```
type varname;
```

e.g. `int x;`

Initialize variables before using them!

Can combine initialization with declaration:

```
int x = 5;
```

Declarations can go anywhere (C99)

C Syntax: Assignment

- Use = sign for assignment
  - set! in Scheme
- The value of an assignment expression is the RHS, while the type is the LHS.

e.g.

```
int x, y;
x = y = 5;
```

Same as `y = 5; x = 5;` (not `x = y`)

C Syntax: True or False?

- Booleans exist in C99, but it is very common to test any type for its "truthiness"
- What evaluates to FALSE in C?
  - 0 (integer)
  - NULL (pointer)
- What evaluates to TRUE in C?
  - everything else…

C syntax : flow control

- Within a function, remarkably close to Java constructs in methods (shows its legacy) in terms of flow control
  - `if-else`
  - `switch`
  - `while` and `for`
  - `do-while`
- Can also use conditional expressions
  - Expressions return VALUES

```
(test) ? then : else;
```

Functions

- Specify return type
  - If no return type, use `void`
- Formal parameters declared after function name
- Function body goes between `{ }`

e.g.

```
int subone(int x) {
    return x - 1;
}
```

C Syntax: `main`

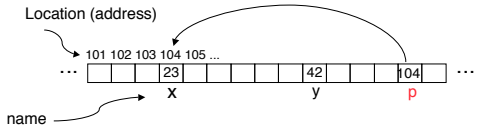
- To get the main function to accept arguments, use this:

```
int main (int argc, char *argv[])
```

- What does this mean?
  - `argc` will contain the number of strings on the command line (the executable counts as one, plus one for each argument).
    - Example: `unix% sort myFile`
  - `argv` is a pointer to an array containing the arguments as strings (more on pointers later).

Pointers

- **Pointer**: A variable that contains the address of another variable.



- A pointer's type includes the number of *'s

```
int *p, **h;
```

  - `p` is type `int *`, `h` is type `int**`

Pointers

- Using pointers
  - `&` operator: get address of a variable
    - returns value whose type has one more *
  - * "dereference operator": two uses
    - RHS: get value from memory pointed at
    - LHS: store value to memory pointed at
    - * also has one more use, in pointer declarations
- Pointers let us change nonlocal variables and keep results
  - Gets around C's pass by copy
  - Don't ever return the address of a local variable!

Pointers & Allocation

- After declaring a pointer:

```
int *ptr;
```

`ptr` doesn't actually point to anything yet. We can either:

  - make it point to something that already exists using &, or
  - allocate room in memory for something new that it will point to with malloc
  - Don't dereference an uninitialized pointer!

Arrays

- Declaration:

```
int ar[size];
```

- Accessing elements:

```
ar[num];
```

- Arrays are (almost) identical to pointers
  - `ar[0]` is the same as `*ar`
  - `ar[2]` is the same as `*(ar+2)`
  - They differ in very subtle ways:
    - Can't increment array variables
    - Can declare filled arrays
    - Using sizeof
- Key Concept: An array variable is like a "pointer" to the first element.

Pointer Arithmetic

- Can do arithmetic on memory address to get a new memory address
- `p+1` returns a ptr to the next array elt.
  - Adds 1*sizeof(arrayelt).
- `*p++` vs `(*p)++` ?
  - `x = *p++` ⇒ `x = *p ; p = p + 1;`
  - `x = (*p)++` ⇒ `x = *p ; *p = *p + 1;`

Pointer Arithmetic

- So what's valid pointer arithmetic?
  - Add an integer to a pointer.
  - Subtract 2 pointers (in the same array).
  - Compare pointers (<, <=, ==, !=, >, >=)
  - Compare pointer to `NULL` (indicates that the pointer points to nothing).
- Everything else is illegal since it makes no sense:
  - adding two pointers
  - multiplying pointers
  - subtract pointer from integer

Improper memory accesses

- **Bus Error**
  - Usually from misaligned address
    - Maybe a freak accident with pointer arithmetic
    - Maybe dereferencing something that wasn't meant to be a pointer
- **Segmentation Fault**
  - When you try to access memory that doesn't belong to you
    - Going out of bounds in an array
    - Invalid pointer values
      - Forgot to initialize
      - Malloc'ing, freeing, then trying to dereference

C Strings

- A string in C is just an array of characters.

```
char string[] = "abc";
```

- How do you tell how long a string is?
  - Last character is followed by a 0 byte (null terminator, '\0')
  - `strlen` counts everything up to and excluding the null byte
  - Make sure to allocate enough space for the null byte!

Pointers to pointers

- You might want an array of pointers.

```
int **int_p_array;
```

- You might want to change a pointer value from within a function

```
void IncrementPtr(int **h)
{   *h = *h + 1;   }
```

Dynamic Memory Allocation

- To allocate room for something new to point to, use `malloc()` (with the help of a typecast and `sizeof`):

```
ptr = (int *) malloc (n*sizeof(int));
```

  - Type on LHS should have one more * than type `malloc`'ed.
- `malloc` returns pointer to memory which contains garbage
- Use `free` on pointers to "unmalloc". Do this or you'll get memory leaks.

C structures

- A `struct` is a data structure composed of existing data types.
  - Commonly used with `typedef`

```
struct point {
    int x;
    int y;
};

typedef struct point point_t;
```

C structures

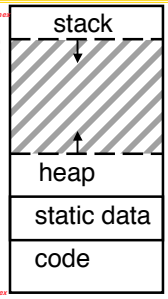
- The C arrow operator (`->`) dereferences and extracts a structure field with a single operator.
- The following are equivalent:

```
struct point *p;
printf("x is %d\n", (*p).x);
printf("x is %d\n", p->x);
```

- Struct size: usually the size of everything inside added up, then word aligned

Normal C Memory Management

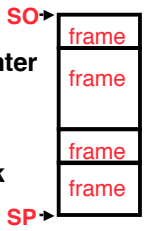
- A program's ***address space*** contains 4 regions:
  - **stack**: local variables, grows downward
  - **heap**: space requested for pointers via `malloc()` ; resizes dynamically, grows upward
  - **static data**: variables declared outside main, does not grow or shrink
  - **code**: loaded when program starts, does not change



For now, OS somehow prevents accesses between stack and heap (gray hash lines). Wait for virtual memory

The Stack

- Stack frame includes:
  - Return "instruction" address
  - Parameters
  - Space for other local variables
- Stack frames contiguous blocks of memory; stack pointer tells where top stack frame is
- When procedure ends, stack frame is tossed off the stack; frees memory for future stack frames



The Heap (Dynamic memory)

- Large pool of memory, not allocated in contiguous order
  - back-to-back requests for heap memory could result blocks very far apart
- In C, tell malloc number of bytes of memory explicitly to allocate item

Memory Management

- How do we manage memory?
- **Code, Static storage are easy**: they never grow or shrink
- **Stack space is also easy**: stack frames are created and destroyed in last-in, first-out (LIFO) order
- **Managing the heap is tricky**: memory can be allocated / deallocated at any time

Heap Management Issues

- Want `malloc()` and `free()` to run quickly.
- Want minimal memory overhead
- Want to avoid ***fragmentation**** – when most of our free memory is in many small chunks
  - In this case, we might have many free bytes but not be able to satisfy a large request since the free bytes are not contiguous in memory.

* This is technically called *external fragmention*

Choosing a free block in `malloc()`

- **Best-fit:** Tries to limit fragmentation but at the cost of time (must examine all free blocks for each malloc). Leaves lots of small blocks (why?)
- **First-fit:** Quicker than best-fit (why?) but potentially more fragmentation. Tends to concentrate small blocks at the beginning of the free list (why?)
- **Next-fit:** Does not concentrate small blocks at front like first-fit, should be faster as a result.

Slab Allocator

- A different way to divide memory
- Divide blocks into "large" and "small" by picking an arbitrary threshold size. Blocks larger than this threshold are managed with a freelist (as before).
- Use a *bitmap* for each range of blocks of the same size.
  - Finding free block = finding 0 bit
  - Freeing block = clearing 1 bit
- Fast for small blocks, no external fragmentation
  - But we have internal fragmentation now

Buddy System

- If no free block of size `n` is available, find a block of size `2n` and split it in to two blocks of size `n`
- When a block of size `n` is freed, if its neighbor (who is its buddy) of size `n` is also free, combine the blocks in to a single block of size `2n`
  - **Buddy** is block in other half larger block

- Helps reduce some of the internal fragmentation in slab allocator

Garbage collection

- Can't do with a weakly typed language because we have no information about object sizes.
- Start with all pointers in global variables and local variables (root set).
- Recursively examine dynamically allocated objects we see a pointer to.
  - These are the only things we need to keep.

Scheme 1: Reference Counting

- Counting pointers.
- For every chunk of dynamically allocated memory, keep a count of number of pointers that point to it.
- When the count reaches 0, reclaim.
- Simple assignment statements can result in a lot of work, since may update reference counts of many items
- Breaks on cyclical structures.

Scheme 2: Mark and Sweep Garbage Col.

- Keep allocating new memory until memory is exhausted, then try to find unused memory.
- Consider objects in heap a graph, chunks of memory (objects) are graph nodes, pointers to memory are graph edges.
  - Edge from A to B ⇒ A stores pointer to B
- Can start with the root set, perform a graph traversal, find all usable memory!
- 2 Phases:
  1. Mark used nodes
  2. Sweep free ones, returning list of free nodes
- Pretty slow because of graph traversal

Scheme 3: Copying Garbage Collection

- Divide memory into two spaces, only one in use at any time.
- When active space is exhausted, traverse the active space, copying all objects to the other space, then make the new space active and continue.
  - Only reachable objects are copied!
- Use "forwarding pointers" to keep consistency
- Bad: Only get to use half the memory.

Assembly Language

- MIPS is an ISA. An ISA is the set of an instructions a CPU can execute.
- MIPS philosophy:
  - RISC. Keep hardware simple and fast.
  - As few instructions as possible
  - Fast common case.

MIPS intro

- In MIPS Assembly Language:
  - Registers replace C variables
    - Registers don't have type, operations do
  - One Instruction (simple operation) per line
  - Simpler is Better
  - Smaller is Faster
- Basic instruction syntax
  - op dest, src1, src2

Data Transfer: Memory and Registers

- Load FROM memory, store TO memory

```
lw $t0,0($s0) # t0 = *s0;
sw $t0,0($s0) # *s0 = t0;
```

- Notes:
  - base register MUST be word aligned if using lw/sw
    - Base register is equivalent to C pointer
  - Base register + offset must be word aligned if using lw/sw

Inequalities in MIPS

- "Set on Less Than"
- Syntax: `slt reg1,reg2,reg3`
  - reg1 = (reg2 < reg3) ? 1 : 0;
  - With bne, get `if(… < …)goto…`
  - With beq, get `if(… >= …)goto…`
- Set greater than?
  - reg1 = (reg2 > reg3) ? 1 : 0;
  - `slt reg1, reg3, reg2`
  - With bne, get `if(… > …)goto…`
  - With beq, get `if(… <= …)goto…`
- Can also just add 1 to operand to do … and equal

MIPS Control flow

- conditional branches

```
beq   register1, register2, L1
bne   register1, register2, L1
```

- unconditional branch

```
j label    # Just like C's goto
```

- Use these to write loops

MIPS Signed vs. Unsigned – diff meanings!

- MIPS Signed v. Unsigned is an "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`)

Calling functions

- `jal` saves next instruction address in $ra and jumps to label

```
jal label
```

- Follow register conventions!
  - CalleE must restore $s0-$s7 and $sp before returning.
  - CalleR must save $ra, $v's, $a's, $t's

MIPS Registers

The constant 0	$0	$zero
Reserved for Assembler	$1	$at
Return Values	$2-$3	$v0-$v1
Arguments	$4-$7	$a0-$a3
Temporary	$8-$15	$t0-$t7
Saved	$16-$23	$s0-$s7
More Temporary	$24-$25	$t8-$t9
Used by Kernel	$26-27	$k0-$k1
Global Pointer	$28	$gp
Stack Pointer	$29	$sp
Frame Pointer	$30	$fp
Return Address	$31	$ra

(From COD 3rd Ed. green insert)
Use names for registers -- code is clearer!

MIPS instruction formats

- MIPS Machine Language Instruction: 32 bits representing a single instruction

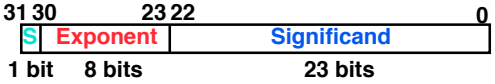
R	opcode	rs	rt	rd	shamt	funct
I	opcode	rs	rt	immediate		
J	opcode	target address				

- Branches use PC-relative addressing, Jumps use absolute addressing.
- Learn to read green sheet!

IEEE 754 Floating Point Standard

- Normal format: $+1.xxxx\ldots xxx_{two} * 2^{yyy\ldots y_{two}}$

Single Precision (DP similar):



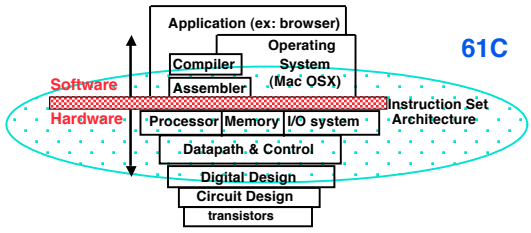
- Sign bit: 1 means negative, 0 means positive
- Significand:
  - To pack more bits, leading 1 implicit for normalized numbers
- Exponent:
  - Get unsigned value, then subtract bias

FP special cases

- Reserve exponents, significands:

Exponent	Significand	Object
0	0	0
0	nonzero	Denorm
1-254	anything	+/- fl. pt. #
255	0	+/- ∞
255	nonzero	NaN

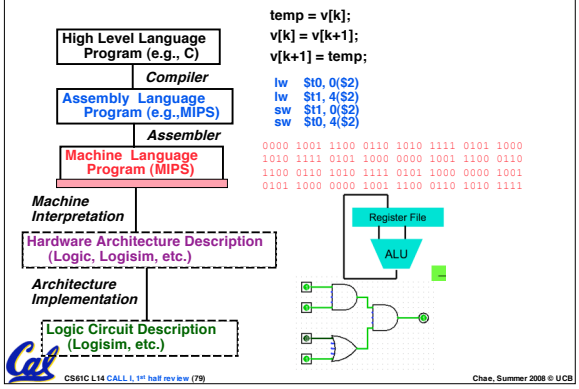
What are "Machine Structures"?



* Coordination of many

levels (layers) of abstraction

61C Levels of Representation



Chae, Summer 2008 © UCB