CS 61C: Great Ideas in Computer Architecture (Machine Structures)

Instructors:
Randy H. Katz
David A. Patterson
http://inst.eecs.Berkeley.edu/~cs61c/fa10

Agenda

- Review
- Real Numbers and Instructions as Numbers
- Assembly Language to Machine Language
- Administrivia
- Technology Break
- More on C and Pointers
- Summary

Review from Last Lecture

- Integer and floating point operations can lead to results too big to store within their representations: overflow/ underflow
- Floating point is an approximation of reals
- Everything is a (binary) number in a computer
  – Instructions and data; stored program concept
- MIPS ISA guided by 4 design principles:
  1. Simplicity favors regularity
  2. Smaller is faster
  3. Make the common case fast
  4. Good design demands good compromises

Goals for Floating Point

- Standard arithmetic for reals for all computers
  – Like two's complement
- Keep as much precision as possible in formats
- Help programmer with errors in real arithmetic
  – +∞, -∞, Not-A-Number (NaN), exponent overflow, exponent underflow
- Keep encoding that is somewhat compatible with two's complement
  – E.g., 0 in Fl. Pt. is 0 in two's complement
  – Make it possible to sort without needing doing floating point comparison

More Floating Point

- Zero: Bit pattern all 0s means 0.000

  ⇒But 0 in exponent should mean most negative exponent (want 0 to be next to smallest real)

  ⇒Can't use two's complement ($1000\ 0000_{two}$)
- *Bias notation*: subtract bias from exponent
  - Single precision uses bias of 127; DP uses 1023
- 0 uses $0000\ 0000_{two}$ => 0-127 = -127;
  ∞, NaN uses $1111\ 1111_{two}$ => 255-127 = +128
  - Smallest SP real can represent: 1.00…00 x 2^{-126}
  - Largest SP real can represent: 1.11…11 x 2^{+127}

9/11/10 Fall 2010 -- Lecture #7 5

MIPS Floating Point Instructions

- C, Java has single precision (**`float`**) and double precision (**`double`**) types
- MIPS instructions: .s for single, .d for double
  - Fl. Pt. Addition single precision: add.s
    Fl. Pt. Addition double precision: add.d
  - Fl. Pt. Subtraction single precision: sub.s
    Fl. Pt. Subtraction double precision: sub.d
  - Fl. Pt. Multiplication single precision: mul.s
    Fl. Pt. Multiplication double precision: mul.d
  - Fl. Pt. Divide single precision: div.s
    Fl. Pt. Divide double precision: div.d

9/11/10 Fall 2010 -- Lecture #7 6

MIPS Floating Point Instructions

- C, Java has single precision (**`float`**) and double precision (**`double`**) types
- MIPS instructions: .s for single, .d for double
  - Fl. Pt. Comparison single precision:
    Fl. Pt. Comparison double precision:
  - Fl. Pt. branch:
- Since rarely mix integers and Fl. Pt., MIPS has separate registers for floating-point operations: \$f0, \$f1, …, \$f31
  - Double precision uses adjacent even-odd pairs of registers:
  - \$f0 and \$f1, \$f2 and \$f3, \$f4 and \$f5, …, \$f30 and \$f31
- Need data transfer instructions for these new registers
  - lwc1 (load word), swc1 (store word)
  - Double precision uses two lwc1 instructions, two swc1 instructions

Encoding of MIPS Instructions: Must Be Unique!

Instruction	Format	op	rs	rt	rd	shamt	funct	address
addu	R	0	reg	reg	reg	0	33_{ten}	*n.a.*
subu	R	0	reg	reg	reg	0	35_{ten}	*n.a.*
sltu	R	0	reg	reg	reg	0	43_{ten}	*n.a.*
sll	R	0	reg	*n.a*	reg	constant	0_{ten}	*n.a.*
addi unsigned	I	9_{ten}	reg	reg	*n.a.*	*n.a.*	*n.a.*	constant
lw (load word)	I	35_{ten}	reg	reg	*n.a.*	*n.a.*	*n.a.*	address
sw (store word)	I	43_{ten}	reg	reg	*n.a.*	*n.a.*	*n.a.*	address
beq	I	4_{ten}	reg	reg	*n.a.*	*n.a.*	*n.a.*	address
bne	I	5_{ten}	reg	reg	*n.a.*	*n.a.*	*n.a.*	address
j (jump)	J	2_{ten}	*n.a.*	*n.a.*	*n.a.*	*n.a.*	*n.a.*	address
jal	J	3_{ten}	*n.a.*	*n.a.*	*n.a.*	*n.a.*	*n.a.*	address
jr (jump reg)	R	0	reg	reg	reg	0	8_{ten}	*n.a.*

Converting C to MIPS Machine code

&A=$t0 (reg 8), $t1 (reg 9), h=$s2 (reg 18)

A[300] = h + A[300];

Format?

lw $t0,1200($t1) _

addu $t1,$s2,$t0 _

sw $t0,1200($t1) _

Instruction	Format	op	rs	rt	rd	shamt	funct	address
addu	R	0	reg	reg	reg	0	33_{ten}	*n.a.*
lw (load word)	I	35_{ten}	reg	reg	*n.a.*	*n.a.*	*n.a.*	address
sw (store word)	I	43_{ten}	reg	reg	*n.a.*	*n.a.*	*n.a.*	address

R-type	op	rs	rt	rd	shamt	funct
I-type	op	rs	rt	address or constant		
J-type	op	address				

Addressing in Branches

I-type	op	rs	rt	address or constant
	6 bits	5 bits	5 bits	16 bits

- Programs much bigger than 2^{16} bytes, but branch address must fit in 16-bit field
  - Must specify a register for branch addresses for big programs: PC = Register + Branch address
  - Which register?
- Conditional branching for IF-statement, loops
  - Tend to be near branches; ½ within 16 instructions
- Idea: *PC-relative branching*

Addressing in Branches

I-type

op	rs	rt	address or constant
6 bits	5 bits	5 bits	16 bits

- Hardware increments PC early, so relative address is PC = (PC + 4) + Branch address
- Another optimization since all MIPS instructions 4 bytes long?
- Multiply value in branch address field by 4!
- MIPS PC-relative branching
  PC = (PC + 4) + (Branch address*4)

Addressing in Jumps

J-type

op	address
6 bits	26 bits

- Same trick for Jumps, Jump and Link
  PC = Jump address * 4
- Since PC = 32 bits, and Jump address * 4 = 28 bits, what about other 4 bits?
- Jump and Jump and Link only changes bottom 28 bits of PC

Converting to MIPS Machine code

Address	Loop:	Format?		
800	sll $t1,$s3,2	_		
804	addu $t1,$t1,$s6	_		
808	lw $t0,0($t1)	_		
812	bne $t0,$s5, Exit	_		
816	addiu $s3,$s3,1	_		
820	j Loop	_		
	Exit:			

R-type	op	rs	rt	rd	shamt	funct
I-type	op	rs	rt	address or constant		
J-type	op	address				

32 bit constants in MIPS

- Can create a 32-bit constant from two 32-bit MIPS instructions
- *Load Upper Immediate (lui or* "Louie") puts 16 bits into upper 16 bits of destination register
- MIPS to load 32-bit constant into register $s0?
  $0000\ 0000\ 0011\ 1101\ 0000\ 1001\ 0000\ 0000_{two}$

lui $s0, 61 # 61 = $0000\ 0000\ 0011\ 1101_{two}$

ori $s0, $s0, 2304 # 2304 = $0000\ 1001\ 0000\ 0000_{two}$

Assembly and Pseudo-instructions

- Turning textual MIPS instructions into machine code called *assembly,* program called *assembler*
  - Calculates addresses, maps register names to numbers, produces binary machine language
  - Textual language called *assembly language*
- Can also accept instructions convenient for programmer but not in hardware
  - *Load immediate* (*li*) allows 32-bit constants, assembler turns into lui + ori (if needed)
  - *Load double* (*ld*) uses two lwc1 instructions to load a pair of 32-bit floating point registers
  - Called *Pseudo-Instructions*

Agenda

- Review
- Real Numbers and Instructions as Numbers
- Administrivia
- Technology Break
- More on C and Pointers
- Summary

Pointers in C

- A pointer is just another kind of value
  - A basic type in C

```
int *ptr;
```

The variable “ptr” is a pointer to an “int”.

Many of Slides 18 to 32 come from “Arrays and Pointers in C” by Alan Cox and T.S. Ng, Rice University, *www.clear.rice.edu/comp221/html/ppt/03-arrays-**pointers**.ppt*

Pointers in C

- If **`T`** is a type, **`T *p`** declares p a pointer to that type
- You can use **`p`** as a pointer to a **`T`**
- You can use **`*p`** as a **`T`**
- **`p++`** increments p by the size of a **`T`**
  - Important because of the way arrays are treated
- You can make a pointer to any variable
  - If **`x`** is any variable, then **`&x`** is its address

Arrays in C

- Array indexing is syntactic sugar for pointers
- **`a[i]`** is treated as **`*(a+i)`**
- To zero out an array:
  - **`for (i=0; i < size; i++) a[i] = 0;`**
  - **`for (i=0; i < size; i++) *(a+i) = 0;`**
  - **`for (p=a; p < a+size; p++) *p = 0;`**

Pointer Operations in C

- Creation
  - & *variable* Returns variable's memory address
- Dereference
  - * *pointer* Returns contents stored at address
- Indirect assignment
  - * *pointer* = *val* Stores value at address
- Of course, still have...Assignment
  - *pointer* = *ptr* Stores pointer in another variable

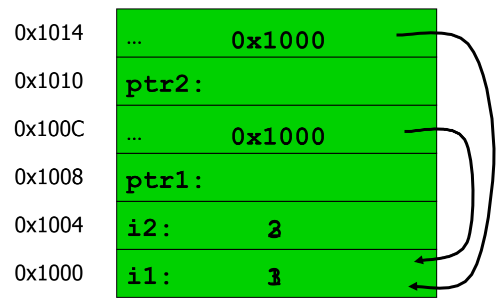
Using Pointers

```
int  i1;
int  i2;
int *ptr1;
int *ptr2;

i1 = 1;
i2 = 2;

ptr1 = &i1;
ptr2 = ptr1;

*ptr1 = 3;
i2 = *ptr2;
```



Using Pointers (cont.)

```
int  int1      = 1036;   /* some data to point to  */
int  int2      = 8;

int *int_ptr1 = &int1;   /* get addresses of data  */
int *int_ptr2 = &int2;

*int_ptr1 = int_ptr2;

*int_ptr1 = int2;
```

What happens?

Type check warning: `int_ptr2` **is not an** `int`

`int1` **becomes 8**

Using Pointers (cont.)

```
int  int1      = 1036;   /* some data to point to  */
int  int2      = 8;

int *int_ptr1 = &int1;   /* get addresses of data  */
int *int_ptr2 = &int2;

int_ptr1 = *int_ptr2;

int_ptr1 = int_ptr2;
```

What happens?

Type check warning: `*int_ptr2` **is not an** `int *`

Changes `int_ptr1` **– doesn't change** `int1`

Pointer Arithmetic

pointer + *number*   *pointer* – *number*

E.g., *pointer* + 1   adds 1 something to a pointer

```
char    *p;
char     a;
char     b;

p = &a;
p += 1;
```

```
int    *p;
int     a;
int     b;

p = &a;
p += 1;
```

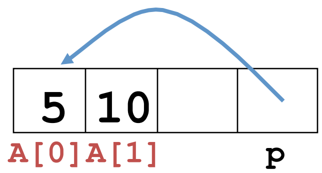
In each, p now points to b
(Assuming compiler doesn't reorder variables in memory)

Adds 1*sizeof(char) to the memory address

Adds 1*sizeof(int) to the memory address

Pointer arithmetic should be used cautiously

Peer Instruction




```
int main(void){
  int A[] = {5,10};
  int *p = A;

  printf("%d %d %d\n", *p,A[0],A[1]);
   p =  p + 1;
  printf("%d %d %d\n", *p,A[0],A[1]);
  *p = *p + 1;
  printf("%d %d %d\n", *p,A[0],A[1]);
  }
```

If the first `printf` outputs <u>5 5 10</u>, what will the other two `printf` output?

```
A (red)     10 5 10       then 11 5 11
B (orange)  11 5 10       then 12 5 10
C)(green)   <other> 5 10 then <3-others>
D)(yellow)  One of 2 printfs causes an ERROR
```

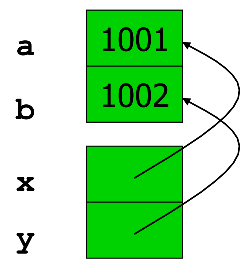
Is Pass by Reference Really by Reference?

- In C, the default passing strategy is pass by copy
- To pass by reference, we use pass by copy – because in C, *everything* is pass by copy
- So, the *value* that we have to pass by copy is the *address* of the actual argument, which we achieve using the *address operator* `&`
- In other words, in C pass by reference is actually pass by copy – because you copy the address

Getting Pass-by-Reference

```
void
set_x_and_y(int *x,
            int *y)
{
   *x = 1001;
   *y = 1002;
}

void
f(void)
{
   int a = 1;
   int b = 2;
   set_x_and_y(&a,&b);
}
```



Arrays and Pointers

•Dirty “secret”:

•Array ≈ pointer to the initial (0th) array element

`a[i]  ≡  *(a+i)`

•An array is passed to a function as a pointer

– The array size is lost!

•Usually bad style to interchange arrays and pointers

– Avoid pointer arithmetic!

Passing arrays:

Really `int *array`

Must explicitly pass the size

```
int
foo(int array[],
    unsigned int size)
{
   … array[size - 1] …
}

int
main(void)
{
   int a[10], b[5];
   … foo(a, 10)… foo(b, 5) …
}
```

Arrays and Pointers

```
int
foo(int array[],
    unsigned int size)
{
   …
   printf("%d\n", sizeof(array));
}

int
main(void)
{
   int a[10], b[5];
   … foo(a, 10)… foo(b, 5) …
   printf("%d\n", sizeof(a));
}
```

What does this print? **8**

... because **array** is really a pointer

What does this print? **40**

Arrays and Pointers

```
int  i;
int  array[10];

for (i = 0; i < 10; i++)
{
  array[i] = …;
}
```

```
int *p;
int  array[10];

for (p = array; p < &array[10]; p++)
{
  *p = …;
}
```

Summary

- Everything is a (binary) number in a computer
  - Instructions and data; stored program concept
- Assemblers can enhance machine instruction set to help assembly-language programmer
- Unlike Java pointers, C pointers you must know
  - when to use a pointer
  - when to *dereference* the pointer
  - when to pass an address to a variable rather than the variable itself
  - when to use pointer arithmetic to change the pointer
  - how to use pointers without making your programs unreadable