

CSE378 - Lecture 3

- Today:
 - Finish up memory
 - Control-flow (branches) in MIPS
 - if/then
 - loops
 - case/switch
 - (maybe) Start: Array Indexing vs. Pointers
 - In particular pointer arithmetic
 - String representation

Quick Review

- Registers x Memory

`lw $t0, 4($a0)`

\$a0 is simply another name for register 4

\$t0 is another name for register _____ (green sheet)

What does \$a0 contain?

What will \$t0 contain after the instruction is executed? (address)

Upper/lower bytes in a register (lui example)

Control flow in high-level languages

- The instructions in a program usually execute one after another, but it's often necessary to alter the normal control flow.
- **Conditional statements** execute only if some test expression is true.

```
// Find the absolute value of a0
v0 = a0;
if (v0 < 0)
    v0 = -v0;           // This might not be executed
v1 = v0 + v0;
```

- **Loops** cause some statements to be executed many times.

```
// Sum the elements of a five-element array a0
v0 = 0;
t0 = 0;
while (t0 < 5) {
    v0 = v0 + a0[t0]; // These statements will
    t0++;             // be executed five times
}
```

Control-flow graphs

```
// Find the absolute value of a0
v0 = a0;
if (v0 < 0)
    v0 = -v0;
v1 = v0 + v0;
```

```
// Sum the elements of a0
v0 = 0;
t0 = 0;
while (t0 < 5) {
    v0 = v0 + a0[t0];
    t0++;
}
```

MIPS control instructions

- MIPS's control-flow instructions

j // for unconditional jumps
bne and **beq** // for conditional branches
slt and **slti** // set if less than (w/o and w an immediate)

- Now we'll talk about
 - MIPS's pseudo branches
 - if/else
 - case/switch

Pseudo-branches

- The MIPS processor only supports two branch instructions, **beq** and **bne**, but to simplify your life the assembler provides the following other branches:

```
blt  $t0, $t1, L1      // Branch if $t0 < $t1
ble  $t0, $t1, L2      // Branch if $t0 <= $t1
bgt  $t0, $t1, L3      // Branch if $t0 > $t1
bge  $t0, $t1, L4      // Branch if $t0 >= $t1
```

- There are also immediate versions of these branches, where the second source is a constant instead of a register.
- Later this quarter we'll see how supporting just **beq** and **bne** simplifies the processor design.

Implementing pseudo-branches

- Most pseudo-branches are implemented using `slt`. For example, a branch-if-less-than instruction `blt $a0, $a1, Label` is translated into the following.

```
slt  $at, $a0, $a1    // $at = 1 if $a0 < $a1
bne  $at, $0, Label   // Branch if $at != 0
```

- This supports immediate branches, which are also pseudo-instructions. For example, `blti $a0, 5, Label` is translated into two instructions.

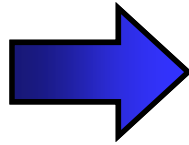
```
slti $at, $a0, 5      // $at = 1 if $a0 < 5
bne  $at, $0, Label   // Branch if $a0 < 5
```

- All of the pseudo-branches need a register to save the result of `slt`, even though it's not needed afterwards.
 - MIPS assemblers use register `$1`, or `$at`, for temporary storage.
 - You should be careful in using `$at` in your own programs, as it may be overwritten by assembler-generated code.

Translating an if-then statement

- We can use branch instructions to translate if-then statements into MIPS assembly code.

```
v0 = a0;  
if (v0 < 0)  
    v0 = -v0;  
v1 = v0 + v0;
```



```
move $v0 $a0  
bge $v0, $0, Label  
sub $v0, 0, $v0  
Label: add $v1, $v0, $v0
```

- Sometimes it's easier to *invert* the original condition.
 - In this case, we changed “continue if $v0 < 0$ ” to “skip if $v0 \geq 0$ ”.
 - This saves a few instructions in the resulting assembly code.

What does this code do?

```
label:  sub    $a0, $a0, 1  
        bne   $a0, $zero, label
```

Loops

```
Loop:      j      Loop          # goto Loop
```

```
for (i = 0; i < 4; i++) {  
    // stuff  
}
```

```
Loop:      add     $t0, $zero, $zero    # i is initialized to 0, $t0 = 0  
           // stuff  
           addi    $t0, $t0, 1         # i ++  
           slti   $t1, $t0, 4         # $t1 = 1 if i < 4  
           bne   $t1, $zero, Loop     # go to Loop if i < 4
```

Control-flow Example

- Let's write a program to count how many bits are set in a 32-bit word.

```
int count = 0;
for (int i = 0 ; i < 32 ; i ++ ) {
    int bit = input & 1;
    if (bit != 0) {
        count ++;
    }
    input = input >> 1;
}
```

```
.text
main:
    li    $a0, 0x1234    ## input = 0x1234
    li    $t0, 0        ## int count = 0;
    li    $t1, 0        ## for (int i = 0

main_loop:
    bge   $t1, 32, main_exit    ## exit loop if i >= 32

    andi  $t2, $a0, 1        ## bit = input & 1
    beq   $t2, $0, main_skip   ## skip if bit == 0

    addi  $t0, $t0, 1        ## count ++

main_skip:
    srl   $a0, $a0, 1        ## input = input >> 1
    add   $t1, $t1, 1        ## i ++

    j     main_loop

main_exit:
    jr    $ra
```

Translating an if-then-else statements

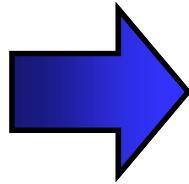
- If there is an **else** clause, it is the target of the conditional branch
 - And the **then** clause needs a jump over the **else** clause

```
// increase the magnitude of v0 by one
```

```
if (v0 < 0)  
    v0 --;
```

```
else
```

```
    v0 ++;  
v1 = v0;
```



```
bge $v0, $0, E
```

```
sub $v0, $v0, 1
```

```
j L
```

```
E: add $v0, $v0, 1
```

```
L: move $v1, $v0
```

- Drawing the control-flow graph can help you out.

Case/Switch Statement

- Many high-level languages support **multi-way branches**, e.g.

```
switch (two_bits) {  
    case 0:    break;  
    case 1:    /* fall through */  
    case 2:    count ++;    break;  
    case 3:    count += 2;  break;  
}
```

- We could just translate the code to if, then, and else:

```
if ((two_bits == 1) || (two_bits == 2)) {  
    count ++;  
} else if (two_bits == 3) {  
    count += 2;  
}
```

- This isn't very efficient if there are many, many **cases**.

Case/Switch Statement

```
switch (two_bits) {  
    case 0:    break;  
    case 1:    /* fall through */  
    case 2:    count ++;    break;  
    case 3:    count += 2;  break;  
}
```

- Alternatively, we can:
 1. Create an array of jump targets
 2. Load the entry indexed by the variable `two_bits`
 3. Jump to that address using the jump register, or `jr`, instruction

Representing strings

- A C-style string is represented by an array of bytes.
 - Elements are one-byte **ASCII codes** for each character.
 - A 0 value marks the end of the array.

32	space	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	\$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(56	8	72	H	88	X	104	h	120	x
41)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	
45	-	61	=	77	M	93]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	del

Null-terminated Strings

- For example, “Harry Potter” can be stored as a 13-byte array.

72	97	114	114	121	32	80	111	116	116	101	114	0
H	a	r	r	y		P	o	t	t	e	r	\0

- Since strings can vary in length, we put a 0, or **null**, at the end of the string.
 - This is called a **null-terminated string**
- Computing string length
 - We’ll look at two ways.

What does this C code do?

```
int foo(char *s) {  
    int L = 0;  
    while (*s++) {  
        ++L;  
    }  
    return L;  
}
```

Array Indexing Implementation of strlen

```
int strlen(char *string) {
    int len = 0;
    while (string[len] != 0) {
        len ++;
    }
    return len;
}
```

Pointers & Pointer Arithmetic

- Many programmers have a vague understanding of pointers
 - Looking at assembly code is useful for their comprehension.
 - (But if you have an aggressive optimizing compiler, you may see the same assembly code for both versions!)

```
int strlen(char *string) {  
    int len = 0;  
    while (string[len] != 0) {  
        len ++;  
    }  
    return len;  
}
```

```
int strlen(char *string) {  
    int len = 0;  
    while (*string != 0) {  
        string ++;  
        len ++;  
    }  
    return len;  
}
```

What is a Pointer?

- A pointer is an address.
- Two pointers that point to the same thing hold the same address
- Dereferencing a pointer means loading from the pointer's address
- In C, a pointer has a type; the type tells us what kind of load to do
 - Use load byte (lb) for char *
 - Use load half (lh) for short *
 - Use load word (lw) for int *
 - Use load single precision floating point (l.s) for float *
- Pointer arithmetic is often used with pointers to arrays
 - Incrementing a pointer (i.e., ++) makes it point to the next element
 - The amount added to the point depends on the type of pointer
 - $pointer = pointer + sizeof(pointer's\ type)$
 - ▶ 1 for char *, 4 for int *, 4 for float *, 8 for double *

What is really going on here...

```
int strlen(char *string) {  
    int len = 0;  
  
    while (*string != 0) {  
        string++;  
        len++;  
    }  
  
    return len;  
}
```

Pointers Summary

- Pointers are just addresses!!
 - “Pointees” are locations in memory
- Pointer arithmetic updates the address held by the pointer
 - “string ++” points to the next element in an array
 - Pointers are typed so address is incremented by sizeof(pointee)