Computer Systems Architecture

http://cs.nott.ac.uk/∼txa/g51csa/

Thorsten Altenkirch and Liyang Hu

School of Computer Science and IT
University of Nottingham

Lecture 05: Comparisons, Loops and Bitwise Operations

Using C

- Java syntax is based on C.
- C is more low-level than Java:
    - Pointers.
    - goto.
- C++ is an extension of C.
- wikibook: *Programming in C* available at http://en.wikibooks.org/wiki/C_Programming

Hello World in C

```
#include <stdio.h>

int main() {
   printf("Hello, world!\n");
}
```

- Store in `hello.c`.
- Compile with:

  `gcc hello.c -o hello`
- Under UNIX run with:

  `hello`

Inequalities

- Previously we learnt `beq` and `bne` (branch on $=$ and $\neq$)
  - Can implement `if(a == b) ...` and `if(a != b) ...`
- But we want other arithmetic comparision operators:

Operator	Name	Abbreviation
$=$	*equals*	`eq`
$\neq$	*not equals*	`ne`
$<$	*less than*	`lt`
$\leq$	*less than or equals*	`le`
$>$	*greater than*	`gt`
$\geq$	*greater than or equals*	`ge`

Comparison Instructions

`slt` *dst*, *src*$_0$, *src*$_1$ – Set on Less Than

- Set *dst* to 1 if *src*$_0$ is less than *src*$_1$, otherwise 0
- `if(`src_0 `<` src_1`)`

      dst `:= 1;`

  `else`

      dst `:= 0;`
- dst `:=` src_0 `<` src_1 `? 1 : 0;`

- There is also `slti` – Set on Less Than Immediate
- Other pseudoinstructions: `seq`, `sne`, `sle`, `sgt`, `sge`, ...

Decisions on Inequalities

- How do we implement `if(a < b) { ... }`, given `if(c != d) { ... }` and `c = a < b ? 1 : 0`?
- We can use two comparisons:
  ```
  c = a < b ? 1 : 0;
  if(c != 0) { ... }
  ```
- Suppose `a` and `b` are `$s0` and `$s1` respectively:
  ```
        slt $t0, $s0, $s1
        beq $t0, $zero, a_ge_b
        # then-block
  a_ge_b:    # rest of program
  ```

Example: Maximum of Two Numbers

- Given two numbers `x` and `y`,
  calculate which is the larger and store it in `m`
- In Java/C: `m = x; if(m < y) m = y;`
- In MIPS assembly, with `$s0`, `$s1` and `$a0` for `x`, `y` and `m`:

```
    move $a0, $s0
    slt $t0, $a0, $s1
    beq $t0, $zero, a0_ge_s1
    move $a0, $s1
a0_ge_s1:
    # Largest number now in $a0
```

A Note on Pseudoinstructions

- The MIPS *processor* only has `slt`, `beq` and `bne` . . .
- But the *assembler* also accepts `sge`, `blt`, `ble` and so on
  - These are *pseudoinstructions* (like `li`, `move`, . . .)
- Pseudoinstructions makes assembly programming easier
  - Write what we mean, not what the processor can do
  - Let the assembler insert the necessary instructions
  - Assembler uses `$at` to implement pseudoinstructions
- In the previous example, we can replace:

```
slt $t0, $a0, $s1
beq $t0, $zero, a0_ge_s1
```

with:

```
bge $a0, $s1, a0_ge_s1
```

While Loops

- Loops are important building blocks in larger programs
- `while` repeats code block as long as condition holds
- What if `i` ≥ 8 before the loop begins?

Java/C

```
while(i < 8) {
    j = j + 3;
    i = i + 1;
}
```

C (using goto)

```
    goto while_cond;
while_loop:
        j = j + 3;
        i = i + 1;
while_cond:
    if(i < 8) goto while_loop;
```

Implementing While Loops

- Rewritten in C using labels and gotos
  - Made our high-level description more concrete
  - Easier to read than assembly instructions
  - Each line has a simple and direct MIPS implementation
- Assume

i	j
$s0	$s1

```
    j while_cond
while_loop:
        addi $s1, $s1, 3
        addi $s0, $s0, 1
while_cond:
    blt $s0, 8, while_loop
```

For Loops

- For loops consist of initialiser, condition and counter parts
- A `for` loop is just a syntactic shortcut for a `while` loop
  - ... but we already know how to implement `while` loops!

For Loop

```
for(i = 0; i < 8; i = i + 1)
    j = j + 3;
```

While Equivalent

```
i = 0;
while(i < 8) {
    j = j + 3;
    i = i + 1;
}
```

The University of Nottingham

Example: How Long is a String?

- Arrive at assembly via a series of translations
- *p means "look up the contents of memory location p"

C (For Loop)

```
length = 0;


for(p = string; *p != 0; p++)
    length++;
```

Example: How Long is a String?

- Arrive at assembly via a series of translations
- *p means "look up the contents of memory location p"

C (While Loop)

```
length = 0;
p = string;

while(*p != 0) {
    length++;
    p++;
}
```

Example: How Long is a String?

- Arrive at assembly via a series of translations
- *p means "look up the contents of memory location p"

C (using goto)

```
    length = 0;
    p = string;
    goto strlen_cond
strlen_loop:
        length++;
        p++;
strlen_cond:
    c = *p;
    if(c != 0) goto strlen_loop;
```

Example: How Long is a String?

- Arrive at assembly via a series of translations
- `length` = `$v0`, `p` = `$a0`, `c` = `$t0`

MIPS Assembly

```
    li $v0, 0
    la $a0, string
    j strlen_cond
strlen_loop:
        addi $v0, $v0, 1
        addi $a0, $a0, 1
strlen_cond:
    lbu $t0, ($a0)
    bne $t0, $zero, strlen_loop
```

Shift to the Left and Shift to the Right

- Shifts move a word's bit pattern to the left or right
- Each shift left (`x = x << 1` in Java syntax)
  - Drops the most significant (leftmost) bit
  - Appends a 0 bit to the least significant end (right)
  - Equivalent to multiplying by 2, ignoring overflow
  - e.g. $0000\,0101_2 \ll 3 = 0010\,1000_2$
- Each shift right (`x = x >> 1` in Java syntax)
  - Drops the least significant (rightmost) bit
  - Prepends a 0 bit to the most significant end (left)
  - Equivalent to dividing by 2, ignoring remainder
  - e.g. $1001\,0011_2 \gg 3 = 0001\,0010_2$

Shift Instructions

sll *dst*, *src*, *shamt* – **shift left logical**

- $dst := src << shamt$

srl *dst*, *src*, *shamt* – **shift right logical**

- $dst := src >> shamt$

Example

Before

$$\$s0 = 7C08\,02A6_{16}$$
$$= 0111\,1100\,0000\,1000\,0000\,0010\,1010\,0110_2$$

```
srl $s0, $s0, 8
sll $s0, $s0, 12
```

After

$$\$s0 = C080\,2000_{16}$$
$$= 1100\,0000\,1000\,0000\,0010\,0000\,0000\,0000_2$$

Bitwise Logical Operations

- Bitwise – no interaction between different bits of a word
- AND (`&`) can be used for testing certain bits of a word
- OR (`|`) can be used for setting certain bits of a word
- XOR (`^`) can be used for inverting certain bits of a word
- NOT (`~`) inverts all the bits in a word

`a`	1100_2
`b`	1010_2
`a & b`	1000_2

`a`	1100_2
`b`	1010_2
`a | b`	1110_2

`a`	1100_2
`b`	1010_2
`a ^ b`	0110_2

The University of Nottingham

Bitwise Logical Instructions

- Immediate variants omitted: `andi`, `ori` and `xori`

`and` dst, src_0, src_1 – Bitwise AND

- dst := src_0 & src_1

`or` dst, src_0, src_1 – Bitwise OR

- dst := src_0 | src_1

`xor` dst, src_0, src_1 – Bitwise XOR

- dst := src_0 ^ src_1

`nor` dst, src_0, src_1 – Bitwise NOR

- dst := ~(src_0 | src_1)
- To get bitwise NOT: `nor` dst, src, `$zero`