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.
#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 `≠`)
- Can implement `if(a == b) ...` and `if(a != b) ...`
- But we want other arithmetic comparison operators:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Name</th>
<th>Abbreviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>equals</td>
<td>eq</td>
</tr>
<tr>
<td>≠</td>
<td>not equals</td>
<td>ne</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
<td>lt</td>
</tr>
<tr>
<td>≤</td>
<td>less than or equals</td>
<td>le</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
<td>gt</td>
</tr>
<tr>
<td>≥</td>
<td>greater than or equals</td>
<td>ge</td>
</tr>
</tbody>
</table>
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
- \[ \text{if}( \text{src}_0 < \text{src}_1) \]
  - **dst** := 1;
  - else
  - **dst** := 0;
- \[ \text{dst} := \text{src}_0 < \text{src}_1 \ ? \ 1 : \ 0; \]

- 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: $m = x; \text{if}(m < y) m = y;$
- 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:

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

with:

```asm
bge $a0, $s1, a0_ge_s1
```
Loops are important building blocks in larger programs

while repeats code block as long as condition holds

What if \( i \geq 8 \) before the loop begins?

**Java/C**

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

**C (using goto)**

```c
    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
\[
\begin{array}{c|c}
  i & j \\
  \hline
  $s0 & $s1
\end{array}
\]

\[j \text{ while}_\text{cond}
\]

\[
\text{while}_\text{loop}:
\begin{align*}
  \text{addi} & \quad $s1, $s1, 3 \\
  \text{addi} & \quad $s0, $s0, 1 \\
\end{align*}
\]

\[
\text{while}_\text{cond}:
\begin{align*}
  \text{blt} & \quad $s0, 8, \text{while}_\text{loop}
\end{align*}
\]
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;
}
```
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)

```c
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
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)**

```c
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
- $\text{length} = v0, p = a0, c = t0$

### 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 0101₂ << 3 = 0010 1000₂
- 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 0011₂ >> 3 = 0001 0010₂
## Shift Instructions

### sll $dst, $src, $shamt – shift left logical

- $dst := $src $<\!< $shamt

### srl $dst, $src, $shamt – shift right logical

- $dst := $src $>> $shamt

### Example

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s0 = 7C0802A6_{16}$</td>
<td>$s0 = C0802000_{16}$</td>
</tr>
<tr>
<td>$= 011111000000100000000010101001102$</td>
<td>$= 11000000100000000010000000000000002$</td>
</tr>
<tr>
<td>srl $s0, $s0, 8</td>
<td>sll $s0, $s0, 12</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>a</th>
<th>1100₂</th>
<th>a</th>
<th>1100₂</th>
<th>a</th>
<th>1100₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>1010₂</td>
<td>b</td>
<td>1010₂</td>
<td>b</td>
<td>1010₂</td>
</tr>
<tr>
<td>a &amp; b</td>
<td>1000₂</td>
<td>a</td>
<td>b</td>
<td>1110₂</td>
<td>a ^ b</td>
</tr>
</tbody>
</table>
Bitwise Logical Instructions

- Immediate variants omitted: andi, ori and xori

**and** \( dst, \ src_0, \ src_1 \) – Bitwise AND
  
  \[ \text{dst} := \text{src}_0 \ \& \ \text{src}_1 \]

**or** \( dst, \ src_0, \ src_1 \) – Bitwise OR
  
  \[ \text{dst} := \text{src}_0 \ \mid \ \text{src}_1 \]

**xor** \( dst, \ src_0, \ src_1 \) – Bitwise XOR
  
  \[ \text{dst} := \text{src}_0 \ ^{\sim} \ \text{src}_1 \]

**nor** \( dst, \ src_0, \ src_1 \) – Bitwise NOR
  
  \[ \text{dst} := \sim (\text{src}_0 \ \mid \ \text{src}_1) \]

To get bitwise NOT: nor \( dst, \ src, \ $zero \)