An artificially intelligent virtual gamer created by computer scientists at The University of Texas at Austin has won the BotPrize by convincing a panel of judges that it was more human-like than half the humans it competed against.

The bots face off in a tournament against one another and about an equal number of humans, with each player trying to score points by eliminating its opponents. Each player also has a "judging gun" in addition to its usual complement of weapons. That gun is used to tag opponents as human or bot.

The winning bots both achieved a humanness rating of 52 percent. Human players received an average humanness rating of only 40 percent.

http://www.sciencedaily.com/releases/2012/09/120926133235.htm
int absdiff(int x, int y) {
    if (x <= y)
        return y-x;
    return x-y;
}

int goto_ad(int x, int y) {
    int result;
    if (x <= y) goto Else;
    result = x-y;
Exit:
    return result;
Else:
    result = y-x;
    goto Exit;
}

int absdiff2(int x, int y) {
    if (x <= y)
        return y-x;
    else
        return x-y;
}
General “Do-While” Translation

C Code

```
do
  Body
while (Test);
```

Goto Version

```
loop:
  Body
  if (Test)
    goto loop
```

- **Body:**
  
  ```
  { 
    Statement_1;
    Statement_2;
    ...
    Statement_n;
  }
  ```

- **Test** returns integer
  
  - = 0 interpreted as false
  - ≠0 interpreted as true
“Do-While” Loop Compilation

```assembly
fact_goto:
   pushl %ebp         # Setup
   movl %esp,%ebp    # Setup
   movl $1,%eax
   movl 8(%ebp),%edx

.L11:
   imull %edx,%eax
   decl %edx
   cmpl $1,%edx
   jg .L11

   movl %ebp,%esp    # Finish
   popl %ebp         # Finish
   ret               # Finish
```

Registers:

- `%edx`  
- `%eax`
**“Do-While” Loop Compilation**

### Goto Version

```c
int
fact_goto(int x)
{
    int result = 1;

    loop:
        result *= x;
        x = x-1;
        if (x > 1)
            goto loop;

    return result;
}
```

### Assembly

```
fact_goto:
    pushl %ebp
    movl %esp,%ebp
    movl $1,%eax
    movl 8(%ebp),%edx

.L11:
    imull %edx,%eax    # result *= x
    decl %edx
    cmpl $1,%edx
    jg .L11            # if > goto loop

    movl %ebp,%esp    # Finish
    popl %ebp
    ret
```

**Registers:**
- `%edx` : `x`
- `%eax` : `result`
“Do-While” Loop Example

C Code

```c
int fact_do(int x)
{
    int result = 1;
    do {
        result *= x;
        x = x-1;
    } while (x > 1);
    return result;
}
```

Goto Version

```c
int fact_goto(int x)
{
    int result = 1;
    loop:
    result *= x;
    x = x-1;
    if (x > 1)
        goto loop;
    return result;
}
```

- Use backward branch to continue looping
- Only take branch when “while” condition holds
“While” Loop Example

C Code

```c
int fact_while(int x) {
    int result = 1;
    while (x > 1) {
        result *= x;
        x = x-1;
    }
    return result;
}
```

Goto Version #1

```c
int fact_while_goto(int x) {
    int result = 1;
    loop:
    if (!(x > 1))
        goto done;
    result *= x;
    x = x-1;
    goto loop;
done:
    return result;
}
```

- Is this code equivalent to the do-while version?
- Must jump out of loop if test fails
Alternative “While” Loop Translation

C Code

```c
int fact_while(int x)
{
    int result = 1;
    while (x > 1) {
        result *= x;
        x = x-1;
    }
    return result;
}
```

Goto Version #2

```c
int fact_while_goto2(int x)
{
    int result = 1;
    if (!(x > 1))
        goto done;
    loop:
    result *= x;
    x = x-1;
    if (x > 1)
        goto loop;
    done:
    return result;
}
```

- Historically used by GCC
- Uses same inner loop as do-while version
- Guards loop entry with extra test
1. %edx, on lines 6 and 11, seems like a new variable. How does it relate to the C code?

2. How is each register used?

3. Annotate the assembly

4. Write a goto version of the C function that mimics the assembly
Arguments: a at %ebp+8, b at %ebp+12
Registers: a in %ecx, b in %ebx, result in %eax, %edx set to apb (a+b)

1. movl 8(%ebp), %ecx  Get a
2. movl 12(%ebp), %ebx  Get b
3. movl $1, %eax  Set result = 1
4. cmpl %ebx, %ecx  Compare a:b
5. jge .L11  If >=, goto done
6. leal (%ebx,%ecx), %edx  Compute apb = a+b
7. movl $1, %eax  Set result = 1

 .L12:
8. imull %edx, %eax  Compute result *= apb
9. addl $1, %ecx  Compute a++
10. addl $1, %edx  Compute apb++
11. cmpl %ecx, %ebx  Compare b:a
12. jg .L12  If >, goto loop

 .L11:
Return result

---

1. int loop_while_goto(int a, int b)
2. {
3.     int result = 1;
4.     if (a >= b)
5.         goto done;
6.     /* apb has same value as a+b in original code */
7.     int apb = a+b;
8.     loop:
9.         result *= apb;
10.        a++;
11.        apb++;
12.     if (b > a)
13.         goto loop;
14.     done:
15.         return result;
16. }
General “While” Translation

While version

while (Test)
  Body

Do-While Version

if (!Test)
  goto done;
do
  Body
while (Test);
done:

Goto Version

if (!Test)
  goto done;
loop:
  Body
if (Test)
  goto loop;
done:
Jump-to-Middle While Translation

C Code

```c
while (Test)
    Body
```

- Avoids duplicating test code
- Unconditional `goto` incurs no performance penalty
- `for` loops compiled in similar fashion

Goto Version

```c
goto middle;
loop:
    Body
middle:
    if (Test)
        goto loop;
```

Goto (Previous) Version

```c
if (!Test)
    goto done;
loop:
    Body
middle:
    if (Test)
        goto loop;
done:
```
Jump-to-Middle Example

```c
int fact_while(int x)
{
    int result = 1;
    while (x > 1) {
        result *= x;
        x--;  
    }
    return result;
}
```

```assembly
# x in %edx, result in %eax
jmp   .L34           # goto Middle
.L35:   # Loop:
imull %edx, %eax   #   result *= x
decl  %edx         #   x-- 
.L34:   # Middle:
cmpl  $1, %edx     #   x:1
jg    .L35        #   if >, goto Loop
```
Implementing Loops

• IA32
  – All loops translated into form based on “do-while”

• x86-64
  – Also make use of “jump to middle”

• Why the difference
  – IA32 compiler developed for machine where all operations costly
  – x86-64 compiler developed for machine where unconditional branches incur (almost) no overhead
“For” Loop Example

C Code

```c
#define WSIZE 8*sizeof(int)
int pcount_for(unsigned x) {
    int i;
    int result = 0;
    for (i = 0; i < WSIZE; i++) {
        unsigned mask = 1 << i;
        result += (x & mask) != 0;
    }
    return result;
}
```

• Is this code equivalent to other versions?
for (Init; Test; Update )

Body

for (i = 0; i < WSIZE; i++) {
    unsigned mask = 1 << i;
    result += (x & mask) != 0;
}

\begin{itemize}
  \item Init
    \begin{itemize}
      \item i = 0
    \end{itemize}
  \item Test
    \begin{itemize}
      \item i < WSIZE
    \end{itemize}
  \item Update
    \begin{itemize}
      \item i++
    \end{itemize}
  \item Body
    \begin{itemize}
      \item unsigned mask = 1 << i;
      \item result += (x & mask) != 0;
    \end{itemize}
\end{itemize}
“For” Loop $\rightarrow$ While Loop

For Version

```c
for (Init; Test; Update) {
  Body
}
```

While Version

```c
Init;
while (Test) {
  Body
  Update;
}
```
“For” Loop $\rightarrow$ ... $\rightarrow$ Goto

**For Version**

```
for (Init; Test; Update)
    Body
```

**While Version**

```
Init;
while (Test) {
    Body
    Update;
}
```

**C++ Code**

```
Init;
if (!Test)
    goto done;
loop:
    Body
    Update
    if (Test)
        goto loop;
    done:
```

```
Init;
    if (!Test)
        goto done;
    if (Test)
        goto loop;
    done:
```
“For” Loop Conversion Example

C Code

```c
#define WSIZE 8*sizeof(int)
int pcoun_for(unsigned x) {
    int i;
    int result = 0;
    for (i = 0; i < WSIZE; i++) {
        unsigned mask = 1 << i;
        result += (x & mask) != 0;
    }
    return result;
}
```

- Initial test can be optimized away

Goto Version

```c
int pcoun_for_gt(unsigned x) {
    int i;
    int result = 0;
    i = 0;
    if (!((i < WSIZE)) goto done;
    loop:
    {
        unsigned mask = 1 << i;
        result += (x & mask) != 0;
    }
    i++;
    if (i < WSIZE) goto loop;
    done:
    return result;
}
```
Using Conditional Moves

• Conditional Move Instructions
  – Instruction supports:
    if (Test) Dest ← Src
  – Supported in post-1995 x86 processors
  – GCC does not always use them
    • Wants to preserve compatibility with ancient processors
    • Enabled for x86-64
    • Use switch --march=686 for IA32

• Why?
  – Branches are very disruptive to instruction flow through pipelines
  – Conditional move do not require control transfer

C Code

```c
val = Test
  ? Then_Expr
  : Else_Expr;
```

Goto Version

```c
tval = Then_Expr;
result = Else_Expr;
t = Test;
if (t) result = tval;
return result;
```
Conditional Move Example: x86-64

```c
int absdiff(int x, int y) {
    int result;
    if (x > y) {
        result = x-y;
    } else {
        result = y-x;
    }
    return result;
}
```

```
absdiff:
x in %edi
movl %edi, %edx
subl %esi, %edx  # tval = x-y
movl %esi, %eax
subl %edi, %eax  # result = y-x
cmpl %esi, %edi  # Compare x:y
cmovg %edx, %eax  # If >, result = tval
ret
y in %esi
```
Bad Cases for Conditional Move

Expensive Computations

```c
val = Test(x) ? Hard1(x) : Hard2(x);
```

- Both values get computed
- Only makes sense when computations are very simple

Risky Computations

```c
val = p ? *p : 0;
```

- Both values get computed
- May have undesirable effects

Computations with side effects

```c
val = x > 0 ? x*=7 : x+=3;
```

- Both values get computed
- Must be side-effect free
Procedure Control Flow

• Use stack to support procedure call and return
• **Procedure call:** `call label`
  – Push return address on stack
  – Jump to `label`

• Return address:
  – Address of instruction beyond `call`
  – Example from disassembly
    
    804854e: e8 3d 06 00 00 00  
    call 8048b90 <main>

    8048553: 50  
    pushl %eax
  – Return address = `0x8048553`

• **Procedure return:** `ret`
  – Pop address from stack
  – Jump to address
Example

```c
yoo (...) {
    •
    •
    who();
    •
    •
}
```

Stack

```
%ebp
%esp
```
Example

```
amI(...) {
    ...
    amI();
    ...
}
```

Stack

- `yoo`
- `who`
- `amI`
- `amI`
- `%ebp` (top)
- `%esp` (bottom)
IA32/Linux Stack Frame

- **Current Stack Frame** ("Top" to Bottom)
  - "Argument build:") Parameters for function about to call
  - Local variables
    - If can’t keep in registers
    - If arrays/structures
    - If & applied to local variable
  - Saved register context
  - Old frame pointer

- **Caller Stack Frame**
  - Return address
  - Pushed by `call` instruction
  - Arguments for this call
Revisiting `swap`

```c
void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

```c
int zip1 = 15213;
int zip2 = 91125;

void call_swap()
{
    swap(&zip1, &zip2);
}
```

Calling `swap` from `call_swap`

```c
call_swap:
    ... ...
    pushl $zip2  # Global Var
    pushl $zip1  # Global Var
    call swap
    ... ...
```

Resulting Stack

```
<table>
<thead>
<tr>
<th>%esp</th>
<th>Rtn adr</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>&amp;zip1</td>
</tr>
<tr>
<td></td>
<td>&amp;zip2</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Calling `swap` from `call_swap`
void swap(int *xp, int *yp) {
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}

swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
    movl 12(%ebp),%ecx
    movl 8(%ebp),%edx
    movl (%ecx),%eax
    movl (%edx),%ebx
    movl %eax,(%edx)
    movl %ebx,(%ecx)
    movl -4(%ebp),%ebx
    movl %esp,%ebp
    popl %ebp
    ret

Set Up

Body

Finish
**swap Setup #1**

**Entering Stack**

- `&zip2`
- `&zip1`
- `Rtn adr`

- `%ebp`
- `%esp`

**Resulting Stack**

- `yp`
- `xp`
- `Rtn adr`
- `Old %ebp`

- `%ebp`
- `%esp`

**swap:**

- `pushl %ebp`
- `movl %esp,%ebp`
- `pushl %ebx`
swap Setup #1

Entering Stack

\[
\begin{array}{c}
\text{%ebp} \\
\text{%esp} \\
&\text{zip2} \\
&\text{zip1} \\
\text{Rtn adr}
\end{array}
\]

\[
\begin{array}{c}
\text{%ebp} \\
\text{%esp} \\
\text{yp} \\
\text{xp} \\
\text{Rtn adr} \\
\text{Old %ebp}
\end{array}
\]

swap:
pushl %ebp
movl %esp,%ebp
pushl %ebx
swap Setup #1

Entering Stack

\[
\begin{align*}
\text{%ebp} & \rightarrow \text{\%ebp} \\
& \text{\&zip2} \\
& \text{\&zip1} \\
& \text{Rtn adr}
\end{align*}
\]

Resulting Stack

\[
\begin{align*}
\text{%ebp} & \rightarrow \text{\%ebp} \\
\text{yp} & \\
\text{xp} & \\
\text{Rtn adr} & \\
\text{Old \%ebp}
\end{align*}
\]

swap:

\[
\begin{align*}
\text{pushl \%ebp} \\
\text{movl \%esp, \%ebp} \\
\text{pushl \%ebx}
\end{align*}
\]
swap Setup #1

Entering Stack

swap:
  pushl %ebp
  movl %esp,%ebp
  pushl %ebx

&zip2
&zip1
Rtn adr

%ebp
%esp

%ebp
%esp

yp
xp
Rtn adr
Old %ebp

%ebp
%esp
### swap Setup #1

**Entering Stack**

- `&zip2` (12)
- `&zip1` (8)
- `Rtn adr` (4)

**Resulting Stack**

- `Rtn adr`
- `Old %ebp`
- `Old %ebx`

---

**Offset relative to `%ebp`**

- `12` for `yp`
- `8` for `xp`
swap Finish #1

swap’s Stack

Resulting Stack

movl -4(%ebp),%ebx
movl %esp,%ebp
popl %ebp
ret

Observation: Saved and restored register %ebx
swap Finish #2

swap's Stack

movl -4(%ebp),%ebx
movl %esp,%ebp
popl %ebp
ret
swap's Stack

- yp
- xp
- Rtn adr
- Old %ebp
- Old %ebx

Resulting Stack

- yp
- xp
- Rtn adr
- Old %ebp

movl  -4(%ebp),%ebx
movl %esp,%ebp
popl %ebp
ret
swap Finish #2

swap’s Stack

movl -4(%ebp),%ebx
movl %esp,%ebp
popl %ebp
ret
swap Finish #3

swap’s Stack

Resulting Stack

movl -4(%ebp),%ebx
movl %esp,%ebp
popl %ebp
ret
swap Finish #4

swap's Stack

movl -4(%ebp),%ebx
movl %esp,%ebp
popl %ebp
ret
swap Finish #4

swap’s Stack

Resulting Stack

• Observation
  - Saved & restored register %ebx
  - Didn’t do so for %eax, %ecx, or %edx
Disassembled **swap**

```
080483a4  <swap>:
  80483a4:  55          push   %ebp
  80483a5:  89 e5       mov    %esp,%ebp
  80483a7:  53          push   %ebx
  80483a8:  8b 55 08    mov    0x8(%ebp),%edx
  80483ab:  8b 4d 0c    mov    0xc(%ebp),%ecx
  80483ae:  8b 1a       mov    (%edx),%ebx
  80483b0:  8b 01       mov    (%ecx),%eax
  80483b2:  89 02       mov    %eax,(%edx)
  80483b4:  89 19       mov    %ebx,(%ecx)
  80483b6:  5b          pop    %ebx
  80483b7:  c9          leave
  80483b8:  c3          ret

Calling Code

8048409:  e8 96 ff ff ff ff    call 80483a4 <swap>
804840e:  8b 45 f8          mov    0xfffffffff8(%ebp),%eax
```
Register Saving Conventions

• When procedure **yoo** calls **who**:
  – **yoo** is the **caller**
  – **who** is the **callee**

• Can Register be used for temporary storage?

  ```
  yoo:
  . . .
  movl $15213, %edx
  call who
  addl %edx, %eax
  . . .
  ret
  
  who:
  . . .
  movl 8(%ebp), %edx
  addl $91125, %edx
  . . .
  ret
  ```

  – Contents of register **%edx** overwritten by **who**
Register Saving Conventions

• When procedure you calls who:
  – you is the caller
  – who is the callee

• Can register be used for temporary storage?

• Conventions
  – “Caller Save”
    • Caller saves temporary in its frame before calling
  – “Callee Save”
    • Callee saves temporary in its frame before using
IA32/Linux Register Usage

• \%eax, \%edx, \%ecx
  – Caller saves prior to call if values are used later

• \%eax
  – also used to return integer value

• \%ebx, \%esi, \%edi
  – Callee saves if wants to use them

• \%esp, \%ebp
  – special
• When is caller save better?
  – What about when calling *multiple* functions?

• Returning floating point vs. returning int
  – Int: %eax (by default – not necessary)
  – Float: x87 register ST0