Lecture for 10/5/09

Miscellaneous final topics in MIPS assembly

How can we perform 64-bit mathematical operations on a 32-bit machine? With multiple instructions...

Adds:

Use base 2^{32} arithmetic to demonstrate how to do 64-bit math with 32-bit operations and registers.

```
  1?
  a0 a1
+ a2 a3
  -----
```

```
// v = a + b in C
unsigned int a_hi, a_lo, b_hi, b_lo;
v_lo = a_lo + b_lo;
carry_bit = (v_lo < a_lo);
v_hi = a_hi + b_hi + carry_bit;

// in 32-bit mips
addu    v1, a1, a3
sltu    t0, v1, a1
addu    v0, a0, a2
addu    v0, v0, t0

// in 64-bit mips
daddu   v0, a0, a1
```

Subtracts:

```
  1?
  a0 a1
- a2 a3
  -----
```

```
// v = a - b in C
unsigned int a_hi, a_lo, b_hi, b_lo;
v_lo = a_lo - b_lo;
carry_bit = (a_lo < v_lo);
v_hi = a_hi - b_hi - carry_bit;

// in 32-bit mips
subu    v1, a1, a3
sltu    t0, a1, v1
subu    v0, a0, a2
subu    v0, v0, t0

// in 64-bit mips
dsubu   v0, a0, a1
```

Multiplies:

```
       a0 a1
     * a2 a3
       -----
     [a1*a3]
  [a0*a3]
+ [a1*a2]
----------
hi = hi(a1*a3) + lo(a0*a3) + lo(a1*a2)
lo = lo(a1*a3)
```

```
// v = a * b in C
unsigned int a_hi, a_lo, b_hi, b_lo;
v_lo = lo(a_lo * b_lo);
v_hi = hi(a_lo * b_lo) + lo(a_lo * b_hi) + lo(a_hi * b_lo);

// in 32-bit mips
multu   a1, a3
mflo    v1
mfhi    v0
multu   a0, a3
mflo    t0
addu    v0, v0, t0
multu   a1, a2
mflo    t0
addu    v0, v0, t0

// in 64-bit mips
dmultu  a0, a1
mflo    v0
```

Divides:

```
// pseudo-code adapted from Art of Assembly Chapter 9:
Quotient := Dividend;
Remainder := 0;
for i:= 1 to 64 do
     Remainder:Quotient := Remainder:Quotient SHL 1;
     if Remainder >= Divisor then
          Remainder := Remainder - Divisor;
          Quotient := Quotient + 1;
     endif
endfor
```

```
// C function to 64-bit divide v = a / b on a 32-bit machine
// remainder is in r_hi and r_lo, but it is not presently returned
// Jeffrey Knockel <jeff250 at unm dot edu> 10/3/09
uint64_t div(uint32_t a_hi, uint32_t a_lo, uint32_t b_hi, uint32_t b_lo) {
    uint32_t v_hi = a_hi, v_lo = a_lo, r_hi = 0, r_lo = 0, r_lo2;
    uint32_t carry_bit;
    int i;
    volatile int dbz;
    if (b_hi | b_lo == 0) { // force divide by zero exception
        dbz = a_lo / b_lo;
    }
    for (i = 64; i > 0; --i) {
        r_hi <<= 1;                          // shift r and shift v into r...
        carry_bit = (0x7fffffffu < r_lo);
        r_hi += carry_bit;
        r_lo <<= 1;
        carry_bit = (0x7fffffffu < v_hi);
        r_lo += carry_bit;
        v_hi <<= 1;
        carry_bit = (0x7fffffffu < v_lo);
        v_hi += carry_bit;
        v_lo <<= 1;

        if (r_hi > b_hi || (r_hi == b_hi && r_lo >= b_lo)) {
            r_lo2 = r_lo - b_lo;
            carry_bit = (r_lo < r_lo2);
            r_lo = r_lo2;
            r_hi = r_hi - b_hi - carry_bit;

            v_lo += 1;
            carry_bit = (v_lo < 1);
            v_hi += carry_bit;
        }
    }

    return ((uint64_t) v_hi << 32) + v_lo;
}

// in 32-bit mips
// see my catalan solution for a special case of the above implemented

// in 64-bit mips
ddivu   a0, a1
mflo    v0
```

Conclusion:

- C compilers can abstract the difficulty of writing instructions for performing 64-bit mathematical operations on a 32-bit architecture, but doing the 64-bit math will still perform poorly, even if the C compiler does the hard work of generating all of the instructions for you.
- Performing 64-bit mathematical operations on a 64-bit architecture takes only a few instructions, but it will still take many more cycles, which is hidden from the programmer and handled by the hardware, although not nearly as many cycles as 64-bit operations on 32-bit machines

Position Independent Code (PIC)

To save memory, we want to load only one copy of a dynamic library into memory for any program that uses it. But, since dynamic libraries can appear in different parts of different programs address spaces, that means that they cannot use absolute addresses and must work regardless of where they are in the address space.

Dynamic linking will bind and resolve symbols among your executable and *.so files while running the program. In the ELF object format, you have a GOT (global offset table) and a PLT (procedure linkage table) to facilitate this. Each linking unit has at least one of these, and a function can calculate the location of these by subtracting the function's address from an immediate offset determined at compile time. In other words, a function always knows where its GOT is relative to itself.

In Linux MIPS, we can calculate the location of the GOT by the following assembler directives at the beginning of a function:

```
    .set noreorder
    .cpload t9
    .set reorder
```

The .cpload directive takes a register as an argument—this is the register that contains the address of this function, and we use this to calculate the address of the GOT and store it in the gp register.

The noreorder/reorder flags are to tell the assembler to not reorder the calculation of the gp register. This is because this calculation must always occur as the first three instructions of a PIC function so that caller functions can jump past these instructions if they can prove that the value in the gp register is already correct for the function that they are calling.

After assembling into a *.o file, the .cpload will be assembled into something like this:

```
00000000 <main>:
   0: 3c1c0000    lui   gp,0x0
                  0: R_MIPS_HI16    _gp_disp
   4: 279c0000    addiu gp,gp,0
                  4: R_MIPS_LO16    _gp_disp
   8: 0399e021    addu  gp,gp,t9
```

The _gp_disp symbol is then resolved by the linker after linking to the location of the GOT relative to the function's address:

```
  400690:   3c1c0002    lui gp,0x2
  400694:   279c8230    addiu   gp,gp,-32208
  400698:   0399e021    addu    gp,gp,t9
```

Consequently, in order for callee functions to have their address in t9, the caller must call them indirectly as follows:

```
    jal     printf
```

Which actually assembles into the following (note that, for PIC code, the address for printf() must be loaded into t9 before jumping there):

```
  24: 8f990000    lw    t9,0(gp)
                  24: R_MIPS_CALL16 printf
  28: 00000000    nop
  2c: 0320f809    jalr  t9
```

Which is resolved to this by the linker:

```
  4006b4:   8f99803c    lw  t9,-32708(gp)
  4006b8:   00000000    nop
  4006bc:   0320f809    jalr    t9
```

The GOT maps symbols into their absolute addresses in memory. For regular data, this mapping is created at link time. For function calls, this mapping is created lazily at call time by the PLT.

The PLT table contains executable code. When we call a dynamically linked function for the first time, the GOT entry for the function will jump us into the PLT. The PLT will jump to the dynamic linker, which will resolve the address for the dynamically linked function and update the respective GOT entry for future calls so that they will jump right into the function proper.

We can use the debugger to see how this works. We will walk through a hello world program that calls printf() twice. We will see that the first time the program attempts to jump to printf(), the GOT entry for printf() will actually jump us into the PLT, which will call the dynamic linker to get the real address for printf() in libc.so and jump there. The second time we call printf(), we will see that the GOT will already be updated by the PLT code from the first call and it will point us to the real printf(), so we will jump right into the printf() in libc.so, avoiding the PLT all together for this and future printf() calls.

```
(gdb) b hello.S:21
Breakpoint 1 at 0x4006a8: file hello.S, line 21.
(gdb) r
Starting program: /home/jeffk/pic/hello

Breakpoint 1, main () at hello.S:21
21        la      a0, hello
Current language:  auto; currently asm
(gdb) si
0x004006ac  21        la      a0, hello
(gdb) si
0x004006b0  21        la      a0, hello
(gdb) si
22        jal     printf
(gdb) si
0x004006b8  22        jal     printf
(gdb) si
0x004006bc  22        jal     printf
(gdb) si
0x00400830 in printf ()
(gdb) x/6i $pc
0x400830 <printf>:      lw    t9,-32752(gp)
0x400834 <printf+4>:    move  t7,ra
0x400838 <printf+8>:    jalr  t9
0x40083c <printf+12>:   li    t8,8
```

```
0x400840 <printf+16>:   nop
0x400844 <printf+20>:   nop
(gdb) si
0x00400834 in printf ()
(gdb) si
0x00400838 in printf ()
(gdb) si
0x2aabf630 in _dl_runtime_resolve () from /lib/ld.so.1
(gdb) b hello.S:23
Breakpoint 2 at 0x4006cc: file hello.S, line 23.
(gdb) c
Continuing.
Hello World

Breakpoint 2, main () at hello.S:24
24        la      a0, hello2
(gdb) si
0x004006d0  24        la      a0, hello2
(gdb) si
0x004006d4  24        la      a0, hello2
(gdb) si
25        jal     printf
(gdb) si
0x004006dc  25        jal     printf
(gdb) si
0x004006e0  25        jal     printf
(gdb) si
0x2ab29e80 in printf () from /lib/libc.so.6
(gdb) c
Continuing.
Hello World2

Program exited normally.
```

Conclusion:

A modern platform that does not support PIC is Windows. Windows' dll's are mapped with a default fixed address in memory that they try to pick such that there will be no other dll loaded there when a program loads them. However, when a program is loaded, there is nothing stopping the dll's that it uses from picking base addresses such that they would overlap others in memory. When this overlap occurs, the loader must rewrite addresses in the overlapping dll's. Moreover, every relocation of a dll cannot use the same memory and must be stored separately.