# LABS

## Week 7

### 1. Conditional Move Instructions

The conventional way to implement conditional operations is through a conditional transfer of ***control***, where the program follows *one execution path when a condition holds* and *another when it does not*. This mechanism is simple and general, but it can be very inefficient on modern processors.

An alternate strategy is through a conditional transfer of data. This approach computes both outcomes of a conditional operation, and then selects one based on whether or not the condition holds. This strategy makes sense only in restricted cases, but it can then be implemented by a simple ***conditional move*** instruction that is better matched to the performance characteristics of modern processors.

We will examine this strategy and its implementation with more recent versions of **IA32** processors.

Starting with the PentiumPro in 1995, recent generations of IA32 processors have had conditional move instructions that either do nothing or copy a value to a register, depending on the values of the condition codes. For years, these instructions have been largely unused. With its default settings, ***gcc*** did not generate code that used them, because that would prevent backward compatibility, even though almost all x86 processors manufactured by Intel and its competitors since 1997 have supported these instructions. More recently, for systems running on processors that are certain to support conditional moves, such as Intel-based Apple Macintosh computers (introduced in 2006) and the 64-bit versions of Linux and Windows, ***gcc*** will generate code using conditional moves. By giving special command-line parameters on other machines, we can indicate to ***gcc*** that the target machine supports conditional move instructions.

As an example, Figure 1.1 shows a variant form of the function ***absdiff*** to illustrate conditional branching.

(a) Original C code

```
1   int absdiff(int x, int y) {
2       return x < y ? y-x : x-y;
3   }
```

(b) Implementation using conditional assignment

```
1   int cmovdiff(int x, int y) {
2       int tval = y-x;
3       int rval = x-y;
4       int test = x < y;
5       /* Line below requires
6          single instruction: */
7       if (test) rval = tval;
8       return rval;
9   }
```

*Figure 1.1 Compilation of conditional statements using conditional assignment*

To understand why code based on conditional data transfers can outperform code based on conditional control transfers, we must understand something about how modern processors operate. As we will see in the next laboratories, processors achieve high performance through ***pipelining***, where an instruction is processed via a sequence of stages, each performing one small portion of the required operations (e.g., fetching the instruction from memory, determining the instruction type, reading from memory, performing an arithmetic operation, writing to memory, and updating the program counter.) This approach achieves high performance by overlapping the steps of the successive instructions, such as fetching one instruction while performing the arithmetic operations for a previous instruction. To do this requires being able to determine the sequence of instructions to be executed well ahead of time in order to keep the pipeline full of instructions to be executed. When the machine encounters a conditional jump (referred to as a "branch"), it often cannot determine yet whether or not the jump will be followed.

Processors employ sophisticated branch prediction logic to try to guess whether or not each jump instruction will be followed. As long as it can guess reliably (modern microprocessor designs try to achieve success rates on the order of 90%), the instruction pipeline will be kept full of instructions. Mispredicting a jump, on the other hand, requires that the processor discard much of the work it has already done on future instructions and then begin filling the pipeline with instructions starting at the correct location. As we will see, such a misprediction can incur a serious penalty, say, 20–40 clock cycles of wasted effort, causing a serious degradation of program performance.

As an example, we ran timings of the ***absdiff*** function on an Intel Core i7 processor using both methods of implementing the conditional operation. In a typical application, the outcome of the test x < y is highly unpredictable, and so even the most sophisticated branch prediction hardware will guess correctly only around 50% of the time. In addition, the computations performed in each of the two code sequences require only a single clock cycle. As a consequence, the branch misprediction penalty dominates the performance of this function. For the IA32code with conditional jumps, we found that the function requires around 13 clock cycles per call when the branching pattern is easily predictable, and around 35 clock cycles per call when the branching

pattern is random. From this we can infer that the branch misprediction penalty is around 44 clock cycles. That means time required by the function ranges between around 13 and 57 cycles, depending on whether or not the branch is predicted correctly.

Figure 1.2 illustrates some of the conditional move instructions added to the IA32 instruction set with the introduction of the PentiumPro microprocessor and supported by most IA32 processors manufactured by Intel and its competitors since 1997. Each of these instructions has two operands: a **source register or memory location S**, and **a destination register R**. As with the different set and jump instructions, the outcome of these instructions depends on the values of the condition codes. The source value is read from either memory or the source register, but it is copied to the destination only if the specified condition holds.

Unlike conditional jumps, the processor can execute conditional move instructions without having to predict the outcome of the test. The processor simply reads the source value (possibly from memory), checks the condition code, and then either updates the destination register or keeps it the same.

| Instruction | | Synonym | Move condition | Description |
|---|---|---|---|---|
| `cmove` | *S*, *R* | `cmovz` | `ZF` | Equal / zero |
| `cmovne` | *S*, *R* | `cmovnz` | `~ZF` | Not equal / not zero |
| `cmovs` | *S*, *R* | | `SF` | Negative |
| `cmovns` | *S*, *R* | | `~SF` | Nonnegative |
| `cmovg` | *S*, *R* | `cmovnle` | `~(SF ^ OF) & ~ZF` | Greater (signed >) |
| `cmovge` | *S*, *R* | `cmovnl` | `~(SF ^ OF)` | Greater or equal (signed >=) |
| `cmovl` | *S*, *R* | `cmovnge` | `SF ^ OF` | Less (signed <) |
| `cmovle` | *S*, *R* | `cmovng` | `(SF ^ OF) \| ZF` | Less or equal (signed <=) |
| `cmova` | *S*, *R* | `cmovnbe` | `~CF & ~ZF` | Above (unsigned >) |
| `cmovae` | *S*, *R* | `cmovnb` | `~CF` | Above or equal (Unsigned >=) |
| `cmovb` | *S*, *R* | `cmovnae` | `CF` | Below (unsigned <) |
| `cmovbe` | *S*, *R* | `cmovna` | `CF \| ZF` | below or equal (unsigned <=) |

*Figure 1.2 The conditional move instructions. Some instructions have 'synonyms', alternate names for the same machine instruction*

To understand how conditional operations can be implemented via conditional data transfers, consider the following general form of conditional expression and assignment:

v = *test-expr* ? *then-expr* : *else-expr*;

With traditional IA32, the compiler generates code having a form shown by the following abstract code:

```
if (!test-expr)
goto false;
v = true-expr;
goto done;
```

false:
v = *else-expr*;
done:

This code contains two code sequences—one evaluating then-expr and one evaluating else-expr. A combination of conditional and unconditional jumps is used to ensure that just one of the sequences is evaluated.

For the code based on conditional move, both the then-expr and the else-expr are evaluated, with the final value chosen based on the evaluation test-expr. This can be described by the following abstract code:

vt = *then-expr*;
v = *else-expr*;
t = *test-expr*;
if (t) v = vt;

Not all conditional expressions can be compiled using conditional moves. Most significantly, the abstract code we have shown evaluates both then-expr and else-expr regardless of the test outcome. If one of those two expressions could possibly generate an error condition or a side effect, this could lead to invalid behavior. As an illustration, consider the following C function:

```
int cread(int *xp)
        {
        return (xp ? *xp : 0);
        }
```

At first, this seems like a good candidate to compile using a conditional move to read the value designated by pointer xp, as shown in the following assembly code:

```
   Invalid implementation of function cread
   xp in register %edx
1      movl    $0, %eax          Set 0 as return value
2      testl   %edx, %edx        Test xp
3      cmovne  (%edx), %eax      if !0, dereference xp to get return value
```

This implementation is invalid, however, since the dereferencing of xp by the ***cmovne*** instruction (line 3) occurs even when the test fails, causing a null pointer dereferencing error. Instead, this code must be compiled using branching code.

## Exercices

### Practice Problem 1

Try to annotate the assembly code.

```
movl 8(%ebp), %ecx
```

```
movl 12(%ebp), %edx

movl %edx, %ebx

subl %ecx, %ebx

movl %ecx, %eax

subl %edx, %eax

cmpl %edx, %ecx

cmovl %ebx, %eax
```

### Practice Problem 2

Compile and assemble the following C code. Is the following code a good candidate to compile using a conditional move? Why?

```
int lcount = 0;

int absdiff_se(int x, int y) {

      return x < y ? (lcount++, y-x) : x-y;

}
```

## 2. Procedures

A procedure call involves passing both data (in the form of procedure parameters and return values) and control from one part of a program to another. In addition, it must allocate space for the local variables of the procedure on entry and deallocate them on exit. Most machines, including IA32, provide only simple instructions for transferring control to and from procedures. The passing of data and the allocation and deallocation of local variables is handled by manipulating the program stack.

IA32 programs make use of the program stack to support procedure calls. The machine uses the stack to pass procedure arguments, to store return information, to save registers for later restoration, and for local storage. The portion of the stack allocated for a single procedure call is called a ***stack frame***.

Figure 2.1 diagrams the general structure of a stack frame. The topmost stack frame is delimited by two pointers, with register **%ebp** serving as the ***frame pointer***, and register **%esp** as the stack pointer. The stack pointer can move while the procedure is executing, and hence most information is accessed relative to the frame pointer.

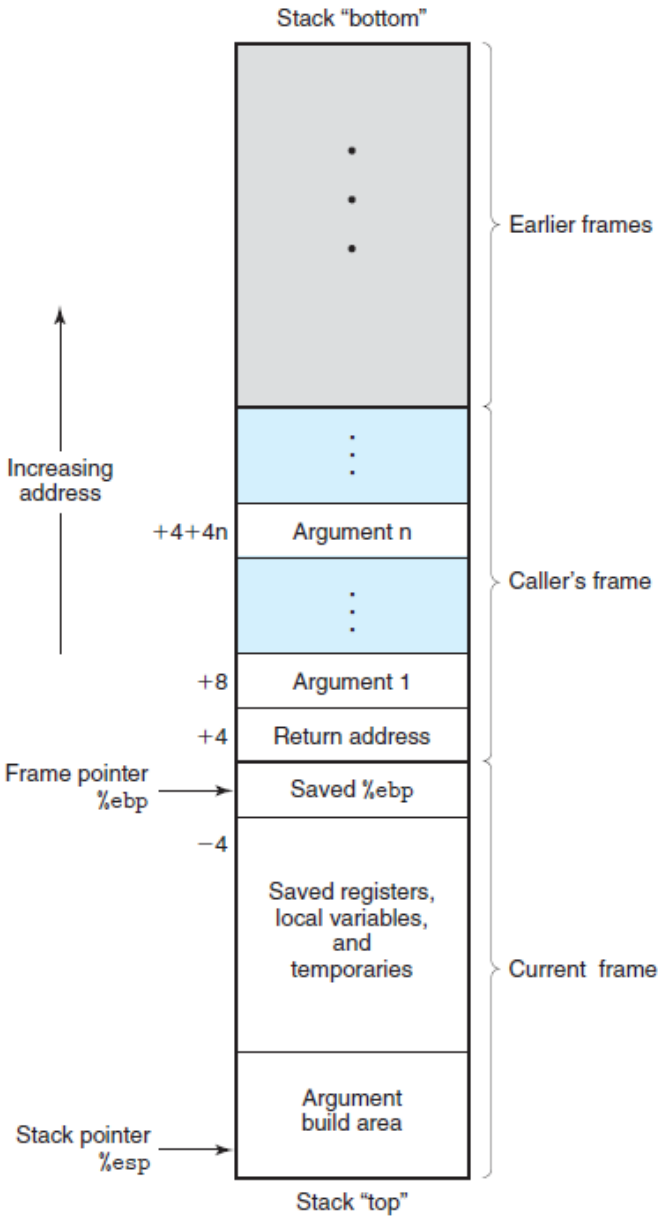



*Figure 2.1 Stack frame structure. The stack is used for passing arguments, for storing return information, for saving registers, and for local storage*

Suppose procedure P (the caller) calls procedure Q (the callee). The arguments to Q are contained within the stack frame for P. In addition, when P calls Q, the return address within P where the program should resume execution when it returns from Q is pushed onto the stack, forming the end of P's stack frame. The stack frame for Q starts with the saved value of the frame pointer (a copy of register %ebp), followed by copies of any other saved register values.

Procedure Q also uses the stack for any local variables that cannot be stored in registers. This can occur for the following reasons:

- There are not enough registers to hold all of the local data.
- Some of the local variables are arrays or structures and hence must be accessed array or structure references.
- The address operator '&' is applied to a local variable, and hence we must be able to generate an address for it.

In addition, Q uses the stack frame for storing arguments to any procedures it calls. As illustrated in Figure 2.1, within the called procedure, the first argument is positioned at offset 8 relative to %ebp, and the remaining arguments (assuming their data types require no more than 4 bytes) are stored in successive 4-byte blocks, so that argument i is at offset 4 + 4i relative to %ebp. Larger arguments (such as structures and larger numeric formats) require larger regions on the stack.

As described earlier, the stack grows toward lower addresses and the stack pointer %esp points to the top element of the stack. Data can be stored on and retrieved from the stack using the pushl and popl instructions. Space for data with no specified initial value can be allocated on the stack by simply decrementing the stack pointer by an appropriate amount. Similarly, space can be deallocated by incrementing the stack pointer.

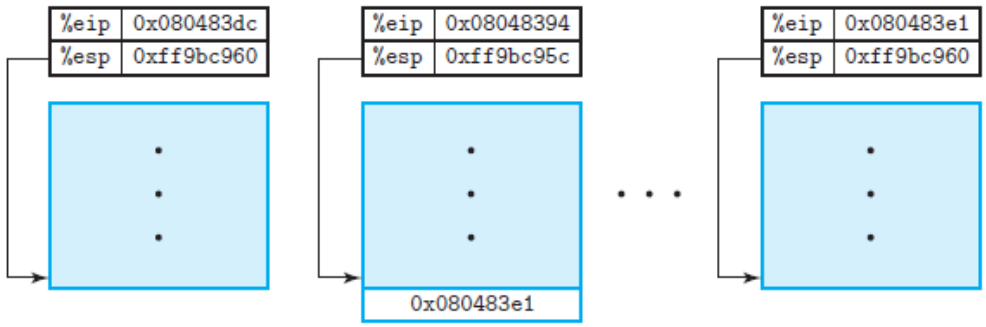
## 3. Transferring Control

The instructions supporting procedure calls and returns are shown in the following table:

| Instruction | | Description |
|---|---|---|
| `call` | *Label* | Procedure call |
| `call` | **Operand* | Procedure call |
| `leave` | | Prepare stack for return |
| `ret` | | Return from call |

The ***call*** instruction has a target indicating the address of the instruction where the called procedure starts. Like jumps, a call can either be direct or indirect. In assembly code, the target of a direct call is given as a label, while the target of an indirect call is given by a *.

```
    Beginning of function sum
1   08048394 <sum>:
2    8048394:  55                    push   %ebp
    . . .
    Return from function sum
3    80483a4:  c3                    ret
    . . .
    Call to sum from main
4    80483dc:  e8 b3 ff ff ff        call   8048394 <sum>
5    80483e1:  83 c4 14              add    $0x14,%esp
```




(a) Executing `call` (b) After `call` (c) After `ret`

## 4. Register Usage Conventions

The set of program registers acts as a single resource shared by all of the procedures. Although only one procedure can be active at a given time, we must make sure that when one procedure (***the caller***) calls another (***the callee***), the callee does not overwrite some register value that the caller planned to use later. For this reason, IA32 adopts a uniform set of conventions for register usage that must be respected by all procedures, including those in program libraries.

By convention, registers %eax, %edx, and %ecx are classified as ***caller-save*** registers.When procedure Q is called by P, it can overwrite these registers without destroying any data required by P. On the other hand, registers %ebx, %esi, and %edi are classified as ***callee-save*** registers. This means that Q must save the values of any of these registers on the stack before overwriting them, and restore them before returning, because P (or some higher-level procedure) may need these values for its future computations. In addition, registers %ebp and %esp must be maintained according to the conventions described here.

As an example, consider the following code:

```
1   int P(int x)
2   {
3       int y = x*x;
4       int z = Q(y);
5       return y + z;
6   }
```

Procedure P computes y before calling Q, but it must also ensure that the value of y is available after Q returns. It can do this by one of two means:

- It can store the value of y in its own stack frame before calling Q; when Q returns, procedure P can then retrieve the value of y from the stack. In other words, P, the caller, saves the value.
- It can store the value of y in a callee-save register. If Q, or any procedure called by Q, wants to use this register, it must save the register value in its stack frame and restore the value before it returns (in other words, the callee saves the value).When Q returns to P, the value of y will be in the callee-save register, either because the register was never altered or because it was saved and restored.

Either convention can be made to work, as long as there is agreement as to which function is responsible for saving which value. IA32 follows both approaches, partitioning the registers into one set that is caller-save, and another set that is callee-save.

## Practice problem 1

The C code function for a recursive factorial has the following code body:

```
int rfact(int n)

{

        int result;

        if (n <= 1)

                result = 1;

        else

                result = n * rfact(n-1);

        return result;

}
```

And the assembly code implementing this function is as follows:

```
rfact:
        pushl %ebp
        movl %esp, %ebp
        pushl %ebx
        subl $4, %esp
        movl 8(%ebp), %ebx
        movl $1, %eax
        cmpl $1, %ebx
        jle .L53
        leal -1(%ebx), %eax
        movl %eax, (%esp)
        call rfact
        imull %ebx, %eax
.L53:
        addl $4, %esp
        popl %ebx
        popl %ebp
        ret
```

Try to annotate the assembly code.

## Practice problem 2

For a C function having the general structure

```
int rfun(unsigned x)
{
        if (..... )
                return ....;
```

```
        unsigned nx = ....;
        int rv = rfun(nx);
        return ....;
}
```

GCC generates the following assembly code (with the setup and completion code omitted):

```
movl 8(%ebp), %ebx
        movl $0, %eax
        testl %ebx, %ebx
        je .L3
        movl %ebx, %eax
        shrl %eax                       ;Shift right by 1
        movl %eax, (%esp)
        call rfun
        movl %ebx, %edx
        andl $1, %edx
        leal (%edx,%eax), %eax
.L3:
```

Fill in the missing expressions in the C code shown above.

## Practice problem 3 (Homework)

Considering that a C function has the following code body:

```
*p = d;
return x-c;
```

And the assembly code implementing this body is as follows:

```
movsbl  12(%ebp),%edx
movl    16(%ebp), %eax
movl    %edx, (%eax)
```

```
movswl   8(%ebp),%eax    ;move sign extend

movl     20(%ebp), %edx

subl     %eax, %edx

movl     %edx, %eax
```

Write a prototype for function **fun**, showing the types and the ordering of the arguments p,d,x and c.