Part 54: Spilling Registers

I've been putting off dealing with register spilling for a while because I knew the issue was going to be thorny. I think what I've done here is a first cut at the problem. It's naive, but it is a start.

The Issues

Registers are a limited commodity in most CPUs. They are the fastest storage units, and we use them to hold temporary results while we evaluate expressions. Once we have stored a result into a more permanent location (e.g. a memory location which represents a variable) we can free the in-use registers and re-use them.

Once we hit expressions of large complexity we run out of enough registers to hold the intermediate results, and this prevents us from evaluating the expression.

At present the compiler can allocate up to four registers. Yes, I know this is a bit artificial; however, there will always be an expression so complex that it can't be evaluated with a fixed number of registers.

Consider this expression, and remember the order of precedence of the C operators:

```
int x= 5 || 6 && 7 | 8 & 9 << 2 + 3 * 4;
```

Each operator on the right has higher precedence that the one on its left. Thus, we need to store 5 into a register, but then evaluate the rest of the expression. Now we store 6 into a register, and ditto. Now, 7 in a register and ditto. Now 8 in a register and ditto.

Oops! We now need to load 9 into a register, but all four registers are allocated. In fact, we'll need to allocate another *four* registers to evaluate this expression. What is the solution?

The solution is to spill registers somewhere in main memory so that we free up a register. However, we also need to reload the spilled register at the point when we need it; this means that it must now be free to have its old value reloaded.

So, we need not only the ability to spill registers somewhere but also track which ones were spilled and when, and reload them as needed. It's tricky. You can see by the external link above that there is a tonne of theory behind optimal register allocation and spilling. This isn't going to be the place for that theory. I'll implement a simple solution and leave you the opportunity to improve the code based on the theory!

Now, where do registers get spilled? We could allocate an arbitrary sized memory heap and store all the spilled registers here. Generally, though, most register spill implementations use the existing stack. Why?

The answers are that we already have hardware-defined *push* and *pop* operations on the stack which are quick. We can (usually) rely on the operating system extending the stack size indefinitely. Also, we divide our stack up into stack frames, one per function. At the end of a function we can simply move the stack pointer, and we don't have to worry about popping off any registers that we spilled and somehow forgot about.

I'm going to use the stack for spilling registers in our compiler. Let's look at the implications of spilling and of using the stack.

The Implications

To do register spilling, we need the ability to:

- Choose and spill one register's value when we need to allocate a register and none are free. It will be pushed on to the stack.
- Reload the spilled register's value fron the stack when we need it.
- Ensure that the register is free at the point when we need to reload its value.
- Before a function call, we need to spill all in-use registers. This is because a function call is an expression. We need to be able to do `2 + 3 * fred(4,5) - 7`, and still have the 2 and 3 in registers once the function returns with its value.

- Thus, we need to reload all the registers that we spilled before a function call.

The above is what we need, regardless of the mechanism. Now let's bring the stack in and see how it will constrain us.

If we can only push a register's value on the stack to spill it, and pop a register's value from the stack, this implies that we have to reload registers in the reverse order in which we spilled them on the stack. Is this something that we can guarantee? In other words, will we ever need to reload a register out of order? If so, the stack isn't going to be the mechanism that we need. Alternatively, can we write our compiler to ensure that the registers reload in reverse spill order?

Some Optimisations

If you have read the external link above, or you know something about register allocation already, then you know there are so many ways we can optimise register allocation and spilling. You probably know much more than I do, so don't giggle too much in the next section.

When we call a function, not all of our registers will be allocated already. Also, some registers will be used to hold some of the argument values for the function. Also, the function will likely return a value and hence destroy a register. Thus, we don't have to spill all of our registers onto the stack before we do a function call. If we were clever, we could work out which registers have to be spilled and only spill these ones.

We can even take a step back and rewrite the AST tree to ease the pressure on our expression evaluation. For example, we could use a form of strength reduction to lower the number of registers allocated.

Consider the expression:

```
2 + (3 + (4 + (5 + (6 + (7 + 8)))))
```

The way it is written, we would have to load 2 into a register, start to evaluate the rest, load 3 into a register and ditto. We would end up with seven register allocations.

However, addition is *commutative*, and therefore we can re-visualise the above expression as:

```
((((2 + 3) + 4) + 5) + 6) + 7
```

Now we can evaluate `2+3` and put it into a register, add on `4` and still only need one register, etc. This is something that the SubC compiler does with its AST trees, and it is something that I'll implement later.

But for now, no optimisations. In fact, the spilling code is going to produce some pretty bad assembly. But at least the assembly that it produces works. Remember, "*premature optimisation is the root of all evil*" -- Donald Knuth.

The Nuts and Bolts

Let's start with the most primitive new functions in `cg.c` :

```
// Push and pop a register on/off the stack
static void pushreg(int r) {
  fprintf(Outfile, "\tpushq\t%s\n", reglist[r]);
}

static void popreg(int r) {
  fprintf(Outfile, "\tpopq\t%s\n", reglist[r]);
}
```

We can use these to spill and reload a register on the stack. Note that I didn't call them `spillreg()` and `reloadreg()` . They are general-purpose and we might use them for something else later.

The `spillreg`

Next up is a new static variable in `cg.c` :

```
static int spillreg=0;
```

This is the next register that we will choose to spill on the stack. Each time we spill a register, we will increment `spillreg` . So it eventually will be 4, then 5, ... then 8, ... then 3002 etc.

Question: why not reset it to zero when we got past the maximum number of registers? The answer is that, when we pop registers from the stack, we need to know when to *stop* popping registers. If we had used modulo arithmetic, we would pop in a fixed cycle and not know when to stop.

That said, we must only spill registers from 0 to `NUMFREEREGS-1`, so we will do some modulo arithmetic in the following code.

Spilling One Register

We spill a register when there are no free registers. We will choose the `spillreg` (modulo NUMFREEREGS) register to spill. In the `alloc_register()` function in `cg.c`:

```
int alloc_register(void) {
  int reg;

  // Try to allocate a register but fail
  ...
  // We have no registers, so we must spill one
  reg= (spillreg % NUMFREEREGS);
  spillreg++;
  fprintf(Outfile, "# spilling reg %d\n", reg);
  pushreg(reg);
  return (reg);
}
```

We choose `spillreg % NUMFREEREGS` as the register to spill, and we `pushreg(reg)` to do so. We increment `spillreg` to be the next register to spill, and we return the newly spilled register number as that is now free. I also have a debug statement in there which I'll remove later.

Reloading One Register

We can only reload a register when a) it becomes free and b) its the most recent register that was spilled onto the stack. Here is where we insert an implicit assumption into our code: we must always reload the most recently-spilled register. We had better make sure that the compiler can keep this promise.

The new code in `free_register()` in `cg.c` is:

```
static void free_register(int reg) {
  ...
  // If this was a spilled register, get it back
  if (spillreg > 0) {
    spillreg--;
    reg= (spillreg % NUMFREEREGS);
    fprintf(Outfile, "# unspilling reg %d\n", reg);
    popreg(reg);
```

```
  } else        // Simply free the in-use register
  ...
}
```

We simply undo the most recent spill, and decrement `spillreg`. Note that this is why we didn't store `spillreg` with a modulo value. Once it hits zero, we know that there are no spilled registers on the stack and there is no point in trying to pop a register value from the stack.

Register Spills Before a Function Call

As I mentioned before, a clever compiler would determine which registers *had* to be spilled before a function call. This is not a clever compiler, and so we have these new functions:

```
// Spill all registers on the stack
void spill_all_regs(void) {
  int i;

  for (i = 0; i < NUMFREEREGS; i++)
    pushreg(i);
}

// Unspill all registers from the stack
static void unspill_all_regs(void) {
  int i;

  for (i = NUMFREEREGS-1; i >= 0; i--)
    popreg(i);
}
```

At this point, while you are either laughing or crying (or both), I'll remind you of a Ken Thompson quote: "*When in doubt, use brute force.*"

Keeping Our Assumptions Intact

We have an implicit assumption built into this code: any reloaded register was the one last spilled. We had better check that this is the case.

For binary expressions, `genAST()` in `gen.c` does this:

```
  // Get the left and right sub-tree values
  leftreg = genAST(n->left, NOLABEL, NOLABEL, NOLABEL, n->op);
  rightreg = genAST(n->right, NOLABEL, NOLABEL, NOLABEL, n->op);
```

```
  switch (n->op) {
    // Do the specific binary operation
  }
```

We allocate the register for the left-hand expression first, then the register for the right-hand expression. If we have to spill registers, then the register for the right-hand expression will be the most recently-spilled register.

Therefore, we had better *free* the register for the right-hand expression first, to ensure that any spilled value will get reloaded back into this register.

I've gone through `cg.c` and made some modifications to the binary expression generators to do this. An example is `cgadd()` in `cg.c`:

```
// Add two registers together and return
// the number of the register with the result
int cgadd(int r1, int r2) {
  fprintf(Outfile, "\taddq\t%s, %s\n", reglist[r2], reglist[r1]);
  free_register(r2);
  return (r1);
}
```

The code used to add into `r2`, free `r1` and return `r2`. Not good, but luckily addition is commutative. We can save the result in either register, so now `r1` returns the result and `r2` is freed. If if was spilled, it will get its old value back.

I *hope* that I've done this everywhere that is needed, and I *hope* that our assumption is not satisfied, but I'm not completely sure yet. We will have to do a lot of testing to be reasonably satisfied.

Changes to Function Calls

We now have the spill/reload nuts and bolts in place. For ordinary register allocations and frees, the above code will spill and reload as required. We also try to ensure that we free the most recently spilled register.

The last thing we need to do is spill the registers before a function call and reload them afterwards. There is a wrinkle: the function may be part of an expression. We need to:

1. Spill the registers first.
2. Copy the arguments to the function (using the registers).

3. Call the function.
4. Reload the registers before we
5. Copy the register's return value.

If we do the last two out of order, we will lose the returned value as we reload all the old registers.

To make the above happen, I've had to share the spill/reload duties between `gen.c` and `cg.c` as follows.

In `gen_funccall()` in `gen.c`:

```
static int gen_funccall(struct ASTnode *n) {
  ...

  // Save the registers before we copy the arguments
  spill_all_regs();

  // Walk the list of arguments and copy them
  ...
  // Call the function, clean up the stack (based on numargs),
  // and return its result
  return (cgcall(n->sym, numargs));
}
```

which does steps 1, 2 and 3: spill, copy, call. And in `cgcall()` in `cg.c`:

```
int cgcall(struct symtable *sym, int numargs) {
  int outr;

  // Call the function
  ...
  // Remove any arguments pushed on the stack
  ...

  // Unspill all the registers
  unspill_all_regs();

  // Get a new register and copy the return value into it
  outr = alloc_register();
  fprintf(Outfile, "\tmovq\t%%rax, %s\n", reglist[outr]);
  return (outr);
}
```

which does the final two steps: reload and copy the return value.

Example Time

Here are some examples which cause register spills: function calls and complex expressions. We'll start with `tests/input136.c`:

```
int add(int x, int y) {
  return(x+y);
}

int main() {
  int result;
  result= 3 * add(2,3) - 5 * add(4,6);
  printf("%d\n", result);
  return(0);
}
```

`add()` needs to be treated as an expression. We put 3 into a register, and spill all the registers before we call `add(2,3)`. We reload the registers before we get the return value. The assembly code is:

```
        movq    $3, %r10        # Get 3 into %r10
        pushq   %r10
        pushq   %r11            # Spill all four registers, thus
        pushq   %r12            # preserving the %r10 value
        pushq   %r13
        movq    $3, %r11        # Copy the 3 and 2 arguments
        movq    %r11, %rsi
        movq    $2, %r11
        movq    %r11, %rdi
        call    add@PLT         # Call add()
        popq    %r13
        popq    %r12            # Reload all four registers, thus
        popq    %r11            # restoring the %r10 value
        popq    %r10
        movq    %rax, %r11      # Get the return value into %r11
        imulq   %r11, %r10      # Multiply 3 * add(2,3)
```

Yes, there is plenty of scope for optimisation here. KISS, though.

In `tests/input137.c`, there is this expression:

```
    x= a + (b + (c + (d + (e + (f + (g + h))))));
```

which requires eight registers and so we'll need to spill four of them. The generated assembly code is:

```
movslq  a(%rip), %r10
movslq  b(%rip), %r11
movslq  c(%rip), %r12
movslq  d(%rip), %r13
pushq   %r10             # spilling %r10
movslq  e(%rip), %r10
pushq   %r11             # spilling %r11
movslq  f(%rip), %r11
pushq   %r12             # spilling %r12
movslq  g(%rip), %r12
pushq   %r13             # spilling %r13
movslq  h(%rip), %r13
addq    %r13, %r12
popq    %r13            # unspilling %r13
addq    %r12, %r11
popq    %r12            # unspilling %r12
addq    %r11, %r10
popq    %r11            # unspilling %r11
addq    %r10, %r13
popq    %r10            # unspilling %r10
addq    %r13, %r12
addq    %r12, %r11
addq    %r11, %r10
movl    %r10d, -4(%rbp)
```

and overall we end up with the correct expression evaluation.

Conclusion and What's Next

Register allocating and spilling is hard to get right, and there is a lot of optimisation theory which can be brought to bear. I've implemented quite a naive approach to register allocating and spilling. It will work but there is substantial room for improvement.

While doing the above, I also fixed the problem with `&&` and `||`. I've decided to write these changes up in the next part, even though the code here already has these changes.
Next step