Writing Assembly (or Hand-Compilation)

Kenjiro Taura

2024/06/16

From high-level programming languages to machine code

- there are *no structured control flows* (for, while, if, etc.); everything must be done by (conditional) jump instructions ($\approx$ “goto” statement)
- an instruction can perform *only a single operation*, so nested expressions (e.g., `a * x + b * y + c * z`) must be broken down into a series of instructions

From high-level programming languages to machine code

- a register $\approx$ a variable, but
  - you have *only a fixed number of them*, so some values may have to be spilled on memory (esp. at function calls)
  - function parameters and return values are on predetermined registers (*calling convention* or *Application Binary Interface*)

Code generation by hand — introspecting "human compiler"

- ex: how to convert the following (which finds $\sqrt{c}$ by the Newton method) into machine language?

```
double sq(double c, long n) {
  double x = c;
  for (long i = 0; i < n; i++) {
    x = x / 2 + c / (x + x);
  }
  return x;
}
```

Step 1 — make all controls “goto”s

```c
double sq(double c, long n) {
  double x = c;
  for (long i = 0; i < n; i++) {
    x = x / 2 + c / (x + x);
  }
  return x;
}
```

⇒

```c
double sq(double c, long n) {
  double x = c;
  long i = 0;
  if (i >= n) goto Lend;
Lstart:
  x = x / 2 + c / (2 * x);
  i++;
  if (i < n) goto Lstart;
Lend:
  return x;
}
```

Step 2 — flatten all nested expressions to “C = A op B”

```c
double sq(double c, long n) {
  double x = c;
  long i = 0;
  if (i >= n) goto Lend;
Lstart:
  x = x / 2 + c / (2 * x);
  i++;
  if (i < n) goto Lstart;
Lend:
  return x;
}
```

$\Rightarrow$

```c
double sq3(double c, long n) {
  double x = c;
  long i = 0;
  if (!(i < n)) goto Lend;
Lstart:
  double t0 = 2;
  double t1 = x / t0;
  double t2 = t0 * x;
  double t3 = c / t2;
  x = t1 + t3;
  i = i + 1;
  if (i < n) goto Lstart;
Lend:
  return x;
}
```

Step 3 — assign “machine variables” (registers or memory) to variables

```
double sq3(double c, long n) { /* c : d0, n : x0 */
  double x = c;         /* x : d1 */
  long i = 0;           /* i : x1 */
  if (!(i < n)) goto Lend;
Lstart:
  double t0 = 2;        /* t0 : d2 */
  double t1 = x / t0; /* t1 : d3 */
  double t2 = t0 * x; /* t2 : d4 */
  double t3 = c / t2; /* t3 : d5 */
  x = t1 + t3;
  i = i + 1;
  if (i < n) goto Lstart;
Lend:
  return x;
}
```

Step 4 — convert them to machine instructions

```
double sq3(double c, long n) {
  /* c : d0, n : x0 */
  # double x = c;        /*x:d1*/
  fmov d1,d0
  # long i = 0;          /*i:x1*/
  mov x1,0
.Lstart:
  # if (!(i < n)) goto Lend;
  cmp x0,x1             /*n - i*/
  ble .Lend
  # double t0 = 2;       /*t0:d2*/
  fmov d2,1.0e2
  # double t1 = x / t0; /*t1:d3*/
  fdiv d3,d1,d2
  # double t2 = t0 * x; /*t2:d4*/
  fmul d4,d2,d1
  # double t3 = c/t2;   /*t3:d5*/
  fdiv d5,d0,d4
  # x = t1 + t3;
  fadd d1,d3,d5
  # i = i + 1;
  add x1,x1,1
  # if (i < n) goto Lstart;
  cmp x0,x1             /* n - i */
  bl .Lstart
.Lend:
  # return x;
  fmov d0,d1
  ret
```

Things are more complex in general…

- we've liberally assigned registers to intermediate results, but:

```
  double x = c;         /* x : d1 */
  long i = 0;           /* i : x1 */
  if (!(i < n)) goto Lend;
Lstart:
  double t0 = 2;        /* t0 : d2 */
  double t1 = x / t0; /* t1 : d3 */
  double t2 = t0 * x; /* t2 : d4 */
  double t3 = c / t2; /* t3 : d5 */
```

- registers are finite (may run out)
- some registers are destroyed (i.e., values on them are lost) across a function call

→ you must use memory ("stack" region) as well

A simplest general strategy for code generation

- in general:
  - there may be too many intermediate results to hold on registers
  - values used after a function call must be saved on memory (or callee-save registers) $\Rightarrow$ *always* using memory (stack) is the simplest strategy
- a register is used only "temporarily" to apply an instruction

A code generation based on the simple strategy

- use the following code (integral) as an example

```
double integ(long n) {
  double x = 0;
  double dx = 1 / (double)n;
  double s = 0;
  for (long i = 0; i < n; i++) {
    s += f(x);
    x += dx;
  }
  return s * dx;
}
```

converting to “goto”s and “C = A op B”s

```
double integ(long n) {
  double x = 0;
  double dx = 1 / (double)n;
  double s = 0;
  for (long i = 0; i < n; i++) {
    s += f(x);
    x += dx;
  }
  return s * dx;
}
```

$\Rightarrow$

```
double integ(long n) {
  double x = 0;
  double t0 = 1;
  double t1 = (double)n;
  double dx = t0 / t1;
  double s = 0;
  long i = 0;
  if (!(i < n)) goto Lend;
Lstart:
  double t2 = f(x);
  s += t2;
  x += dx;
  i += 1;
  if (i < n) goto Lstart;
Lend:
  double t3 = s * dx;
  return t3;
}
```

allocate memory slot for intermediate values

```
double integ(long n) {      /*  n : sp+16 */
  double x = 0;             /*  x : sp+24 */
  double t0 = 1;            /* t0 : sp+32 */
  double t1 = (double)n;    /* t1 : sp+40 */
  double dx = t0 / t1;      /* dx : sp+48 */
  double s = 0;             /*  s : sp+56 */
  long i = 0;               /*  i : sp+64 */
  if (!(i < n)) goto Lend;
Lstart:
  double t2 = f(x);         /* t2 : sp+72 */
  s += t2;
  x += dx;
  i += 1;
  if (i < n) goto Lstart;
Lend:
  double t3 = s * dx;       /* t3 : sp+80 */
  return t3;
}
```

Generate instructions

```
double integ(long n) { /*    n: sp+16*/
  stp x29,x30,[sp,80]!
  mov x29,sp
  str x0,[sp,16]
  /* double x = 0;           x: sp+24*/
  movi d0,#0
  str d0,[sp+24]
  /* double t0 = 1;         t0: sp+32*/
  fmov d0,1.0e+0
  str d0,[sp,32]
  /* double t1 = (double)n; t1: sp+40*/
  ldr x0,[sp,16]
  scvtf d0,x0
  str d0,[sp,40]
  /* double dx = t0 / t1;   dx: sp+48*/
  ldr d0,[sp,32]
```

```
ldr d1,[sp,40]
fdiv d0,d0,d1
str d0,[sp,48]
/* double s = 0;                s: sp+56*/
movi d0,#0
str d0,[sp,56]
/* long i = 0;                  i: sp+64*/
mov x0,#0
str x0,[sp,64]
/* if (!(i < n)) goto Lend; */
ldr x0,[sp,64]
ldr x1,[sp,16]
cmp x0,x1   // i - n
bge Lend
```

Generate instructions

```
Lstart:
  /* double t2 = f(x);    t2: sp+72*/
  ldr d0,[sp,24]
  bl f
  str d0,[sp,72]
  /* s += t2; */
  ldr d0,[sp,56]
  ldr d1,[sp,72]
  add d0,d0,d1
  str d0,[sp,56]
  /* x += dx; */
  ldr d0,[sp,24]
  ldr d1,[sp,48]
  add d0,d0,d1
  str d0,[sp,24]
  /* i += 1; */
  ldr d0,[sp,64]
  add d0,d0,1
  str d0,[sp,64]
  /* if (i < n) goto Lstart; */
  ldr x0,[sp,64]
  ldr x1,[sp,16]
  cmp x0,x1   // i - n
  bl Lstart
Lend:
  /* double t3 = s * dx;    t3: sp+80*/
  ldr d0,[sp,56]
  ldr d1,[sp,48]
  mul d0,d0,d1
  str d0,[sp,80]
  /* return t3; */
  ldr d0,[sp,80]
  ret
}
```