CSE 401 – Compilers

Lecture 17: Code Generation for Basic Language Constructs

Michael Ringenburg
Winter 2013

Reminders

- Project part 3 was assigned last week
  - Due Friday, March 1
- Part 4 will be due on Friday, March 15 (last day of class). I will put the assignment out next week – likely before part 3 is due, in case anyone wants to get a head start.
  - The next few lectures will cover the material you need for part 4.

Agenda

- Review of the example we rushed through at the end of class last Wednesday.
- Talk about code generation for basic constructs
- Next time: code generation for OO constructs
- Next week: project code generation (how to apply this week's material for your project)

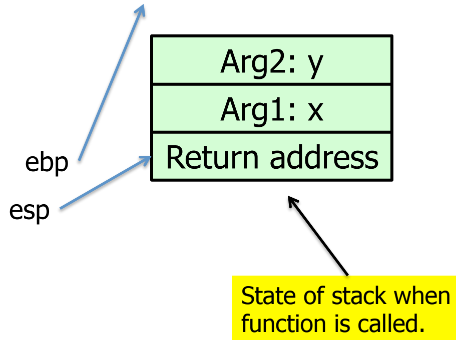
Review: Assembly for an Example Function

- Source code

```
int sumOf(int x, int y) {
   int a, int b;
   a = x;
   b = a + y;
   return b;
}
```

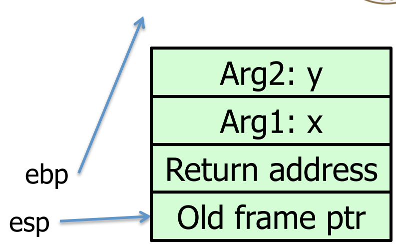
Review: Assembly for an Example Function

```
;; int sumOf(int x, int y) {
;;   int a, int b;
sumOf:
    ;; prologue
    push  ebp
    mov   ebp,esp
    sub    esp, 8

;;   a = x;
    mov  eax,[ebp+8]
    mov  [ebp-4],eax
```



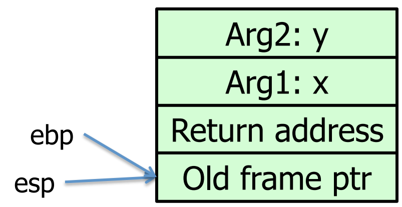
Review: Assembly for an Example Function

```
;; int sumOf(int x, int y) {
;;   int a, int b;
sumOf:
    ;; prologue
    push  ebp   ; store old base
    mov   ebp,esp
    sub    esp, 8

;;   a = x;
    mov  eax,[ebp+8]
    mov  [ebp-4],eax
```



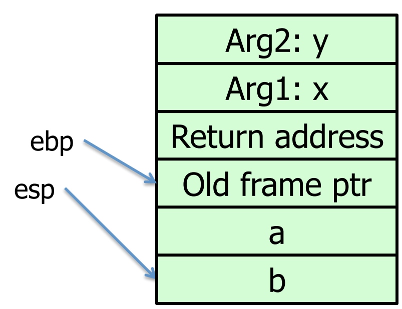
Review: Assembly for an Example Function

```
;; int sumOf(int x, int y) {
;;   int a, int b;
sumOf:
    ;; prologue
    push  ebp
    mov   ebp,esp    ; new base
    sub   esp, 8

;;   a = x;
    mov  eax,[ebp+8]
    mov  [ebp-4],eax
```



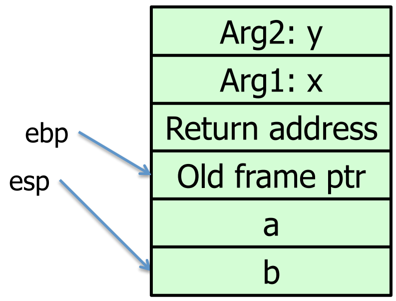
Review: Assembly for an Example Function

```
;; int sumOf(int x, int y) {
;;   int a, int b;
sumOf:
    ;; prologue
    push  ebp
    mov   ebp,esp
    sub   esp, 8       ; allocate
                       ; stack frame
;;   a = x;
    mov  eax,[ebp+8]
    mov  [ebp-4],eax
```



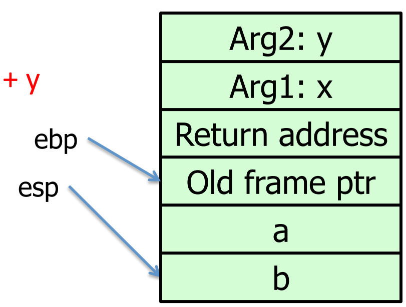
Review: Assembly for an Example Function

```
;; int sumOf(int x, int y) {
;;  int a, int b;
sumOf:
    ;; prologue
    push  ebp
    mov   ebp,esp
    sub    esp, 8

;;  a = x;
    mov  eax,[ebp+8]   ; eax <- x
    mov  [ebp-4],eax    ; a <- x
```



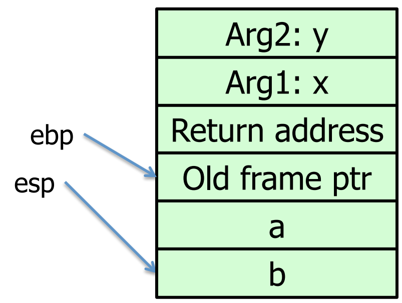
Review: Assembly for an Example Function

```
;; b = a + y;
    mov  eax,[ebp-4]    ; eax <- a
    add   eax,[ebp+12] ; eax <- a + y
    mov  [ebp-8],eax    ; b <- a+y

;; return b;
    mov  eax,[ebp-8]

;; epilogue
    mov  esp,ebp
    pop   ebp
    ret
;; }
```



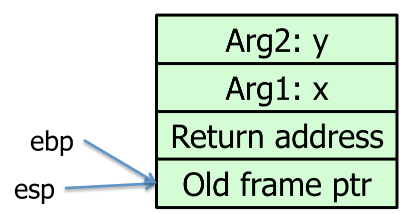
Review: Assembly for an Example Function

```
;; b = a + y;
    mov  eax,[ebp-4]
    add   eax,[ebp+12]
    mov  [ebp-8],eax

;; return b;
    mov  eax,[ebp-8] ; eax <-b

;; epilogue
    mov  esp,ebp
    pop   ebp
    ret
;; }
```



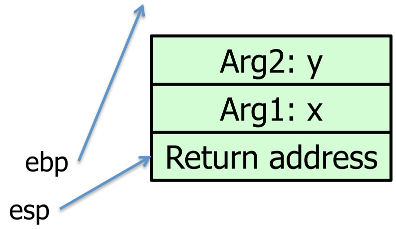
Review: Assembly for an Example Function


```
;; b = a + y;
    mov  eax,[ebp-4]
    add   eax,[ebp+12]
    mov  [ebp-8],eax

;; return b;
    mov  eax,[ebp-8]

;; epilogue
    mov  esp,ebp ; pop frame
    pop   ebp
    ret
;; }
```



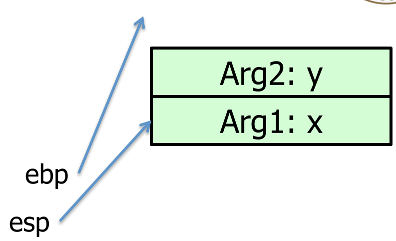
Review: Assembly for an Example Function

```
;; b = a + y;
    mov  eax,[ebp-4]
    add  eax,[ebp+12]
    mov  [ebp-8],eax

;; return b;
    mov  eax,[ebp-8]

;; epilogue
    mov  esp,ebp
    pop  ebp ; restore old base
    ret
;; }
```



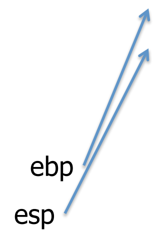
Review: Assembly for an Example Function

```
;; b = a + y;
    mov  eax,[ebp-4]
    add  eax,[ebp+12]
    mov  [ebp-8],eax

;; return b;
    mov  eax,[ebp-8]

;; epilogue
    mov  esp,ebp
    pop  ebp
    ret  ; eip <- return address (and pop RA)
;; }
```



Review: Assembly for an Example Function

```
;; b = a + y;
   mov  eax,[ebp-4]
   add  eax,[ebp+12]
   mov  [ebp-8],eax

;; return b;
   mov  eax,[ebp-8]

;; epilogue
   mov  esp,ebp
   pop  ebp
   ret
;; }
```



Caller then pops arguments and stores return value from eax.

Basic Code Generation Strategy

- Walk the IR (for us, an AST), outputting code for each construct encountered
- Handling of node's children is dependent on type of node
  - E.g., for binary operation like +:
    - Generate code to compute operand 1 (and store result)
    - Generate code to compute operand 2 (and store result)
    - Generate code to load operand results and add them together

Conventions for Examples

- The following slides will walk through how this is done for many common language constructs
- Examples show code snippets in isolation
  - Much the way we'll generate code for different parts of the AST in our compilers
- Register eax used below as a generic example
  - Rename as needed for more complex code using multiple registers
- A few *peephole optimizations* included below for a flavor of what's possible
  - Localized optimizations performed on small ASM instruction sequences.

Variables

- For our purposes, assume all data will be in either:
  - A stack frame (method local variables)
  - An object (instance variables)
- Local variables accessed via ebp

```
mov  eax,[ebp+12]
```

- Object instance variables accessed via an object address in a register
  - Details later

What we're skipping for now

- Real code generator needs to deal with many things like:
  - Which registers are busy at which point in the program
  - Which registers to spill into memory when a new register is needed and no free ones are available
    - (x86: temporaries are often pushed on the stack, but can also be stored at preallocated locations in the stack frame)
  - Exploiting the full instruction set
- Later we'll present a very simple strategy for dealing with these issues in your project.

Code Generation for Constants

- Source

```
17
```

- x86

```
mov   eax,17
```

  - Idea: realize constant value in a register

- Optimization: if constant is 0

```
xor   eax,eax
```

  - Smaller and faster

Assignment Statement

- Source

  var = exp;

- x86

  <code to evaluate exp into, say, eax>

  mov [ebp+offset$_{var}$],eax

Unary Minus

- Source

  -exp

- x86

  <code evaluating exp into eax>

  neg eax

- Optimization
  - Collapse -(-exp) to exp
- Unary plus is a no-op

Binary +

- Source

  exp1 + exp2

- x86

  <code evaluating exp1 into eax>

  <code evaluating exp2 into edx>

  add eax,edx

Binary +

- Optimizations
  - If exp2 is a simple variable or constant, don't need to load it into another register first. Instead:

    add eax,imm_{Const} ; imm is constant

    add eax,[ebp+$offset_{var}$] ; offset is variable's stack offset
  - Change exp1 + (-exp2) into exp1-exp2
  - If exp2 is 1

    inc eax

    - Somewhat surprising: whether this is better than add eax,1 depends on processor implementation and has changed over time

Binary -, *

- Same as +
  - Use sub for –
  - Use imul for *
- Optimizations
  - Use left shift to multiply by powers of 2
  - If your multiplier is slow (or busy), you can do 10*x = (8*x)+(2*x) (2 shifts and an add)
  - Use x+x instead of 2*x, etc. (often faster)
  - Can use lea eax,[eax + eax*4] to compute 5*x, then add eax,eax to get 10*x, etc. etc.
  - Use dec for x-1

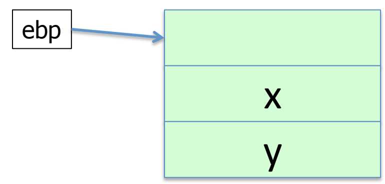
Integer Division

- Ghastly on x86
  - Only works on 64 bit int divided by 32-bit int
  - Requires use of specific registers
- Source

  exp1 / exp2
- X86 (assuming exp1 and exp2 are 32 bit operands)

  <code evaluating exp1 into eax **ONLY**>
  <code evaluating exp2 into ebx>
  cdq ; extend to edx:eax, clobbers edx
  idiv ebx ; quotient in eax; remainder in edx

Example: 5*x + 4/y



Control Flow

- Basic idea: decompose higher level operation into conditional and unconditional gotos
- In the following, j_{false} is used to mean jump when a condition is false
  - No such instruction on x86
  - Will have to realize with appropriate sequence of instructions to set condition codes followed by conditional jumps – we'll discuss later.
  - Normally don't actually generate the value "true" or "false" in a register

While

- Source

    while (cond) stmt

- X86

```
test:   <code evaluating cond>
        j_false done
        <code for stmt>
        jmp  test
done:
```

  – Note: In generated asm code we'll need to generate *unique* labels for each loop, conditional statement, etc.

Optimization for While

- Put the test at the end

```
        jmp  test
loop:   <code for stmt>
test:   <code evaluating cond>
        j_true  loop
```

- Why bother?
  - Pulls one instruction (jmp) out of the loop
  - Older processors: may avoid a pipeline stall on jmp on each iteration
    - Although modern processors will often predict control flow and avoid the stall – x86 does this particularly well

Do-While

- Source

  do stmt while(cond);

- x86

  loop: <code for stmt>

  <code evaluating cond>

  j_{true} loop

If

- Source

  if (cond) stmt

- x86

  <code evaluating cond>

  j_{false} skip

  <code for stmt>

  skip:

If-Else

- Source

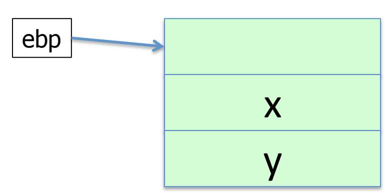
    if (cond) stmt1 else stmt2

- x86

```
        <code evaluating cond>
        j_false else
        <code for stmt1>
        jmp  done
else:   <code for stmt2>
done:
```


Example

```
if (y > 0)
  x--;
while (x > 0) {
  y++;
  x--;
}
```



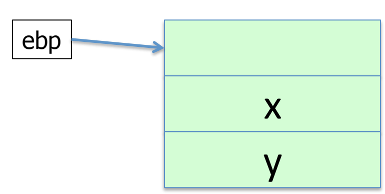
Jump Chaining

- Observation: naïve implementation can produce jumps to jumps
  - Like in previous example!
- Optimization: if a jump has as its target an unconditional jump, change the target of the first jump to the target of the second
  - Repeat until no further changes
  - Often done in peephole optimization pass after initial code generation

Example, revisited

```
       cmp [ebp-8],0
       jng skip
       dec [ebp-4]
skip:  jmp test
loop:  inc [ebp-8]
       dec [ebp-4]
test:  cmp [ebp-4],0
       jg loop
```

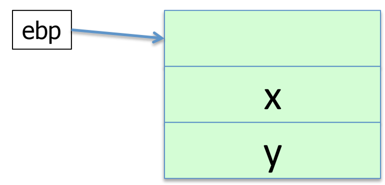
```
if (y > 0)
  x--;
while (x > 0) {
  y++;
  x--;
}
```



Example, revisited

        cmp [ebp-8],0
        jng ~~skip~~ test
        dec [ebp-4]
skip: **jmp test**
loop: inc [ebp-8]
        dec [ebp-4]
test: cmp [ebp-4],0
        jg loop

```
if (y > 0)
  x--;
while (x > 0) {
  y++;
  x--;
}
```



Boolean Expressions

- What do we do with this?
  x > y
- It is an expression that evaluates to true or false
  - Could generate the value (0/1 or whatever the local convention is)
  - But normally we don't want/need the value; we're only trying to decide whether to jump
    - One exception: assignment expressions, e.g.,
      while (my_bool = (x < y)) { … }

Code for exp1 > exp2

- Generated code depends on context
  - What is the jump target?
  - Jump if the condition is true or if false?
- Example: evaluate exp1 > exp2, jump on false, target if jump taken is L123

```
<evaluate exp1 to eax>
<evaluate exp2 to edx>
cmp  eax,edx
jng   L123    ; greater-than test, jump on false, so jng
              ; (jump not greater)
```


Boolean Operators: !

- Source

```
! exp
```

- Context: evaluate exp and jump to L123 if false (or true)
- To compile !, compile the exp test, and reverse the jump conditional
  - E.g., jg → jng, jng → jg

Boolean Operators: && and ||

- In C/C++/Java/C#, these are *short-circuit* operators
  - Right operand is evaluated only if needed
- Basically, evaluate left operand, insert conditional jump based on short-circuit condition (left operand false → && is false, left operand true → || is true).

Example: Code for &&

- Source

```
if (exp1 && exp2) stmt
```

- x86

```
    <code for exp1>
    j_false skip
    <code for exp2>
    j_false skip
    <code for stmt>
skip:
```

Example: Code for ||

- Source

  if (exp1 || exp2) stmt

- x86

```
          <code for exp1>
          j_true doit
          <code for exp2>
          j_false skip
doit:     <code for stmt>
skip:
```


Realizing Boolean Values

- If a boolean value needs to be stored in a variable or method call parameter, generate code needed to actually produce it
- Typical representations: 0 for false, +1 or -1 for true
  - C specifies 0 and 1; we'll use that
  - Best choice can depend on machine instructions; normally some convention is established during the primeval history of the architecture

Boolean Values: Example

- Source

  var = bexp ;
- x86

  <code for bexp>
  j_{false} genFalse
  mov eax,1
  jmp storeIt
  genFalse:
  mov eax,0
  storeIt:mov [ebp+$offset_{var}$],eax ; generated by assign

Better, If Enough Registers

- Source

  var = bexp ;
- x86

  xor eax,eax
  <code for bexp>
  j_{false} storeIt
  inc eax
  storeIt:mov [ebp+$offset_{var}$],eax ; generated by assign

  - Fewer jumps, and xor and inc are cheaper/smaller than mov
  - Even better: use conditional move instruction to avoid jump
    - If available
  - Can also use conditional move instruction for sequences like x = y<z ? y : z

Other Control Flow: switch

- Naïve: generate a chain of nested if-else if statements
- Better: switch is designed to allow an O(1) selection in usual case, provided the set of switch values is reasonably compact
- Idea: create a 1-D array of jumps or labels and use the switch expression to select the right one
  - Need to generate the equivalent of an if statement to ensure that expression value is within bounds

Switch

- Source

```
switch (exp) {
  case 0: stmts0;
  case 1: stmts1;
  case 2: stmts2;
}
```

- X86

```
      <put exp in eax>
      "if (eax < 0 || eax > 2)
         jmp defaultLabel"
      mov eax,swtab[eax*4]
      jmp eax
      .data
      swtab    dd L0
               dd L1
               dd L2
      .code
L0:   <stmts0>
L1:   <stmts1>
L2:   <stmts2>
```

Arrays

- Several variations
- C/C++/Java
  - 0-origin; an array with n elements contains variables a[0]…a[n-1]
  - 1 or more dimensions; row major order
- Key step is to evaluate a subscript expression and calculate the location of the corresponding element

0-Origin 1-D Integer Arrays

- Source

  exp1[exp2]

- x86

  <evaluate exp1 (array address) in eax>

  <evaluate exp2 in edx>

  address is [eax+4*edx] ; assumes 4 bytes
  ; per element

2-D Arrays

- C, etc. use row-major order
  - E.g., an array with 3 rows and 2 columns is stored in this sequence: a(0,0), a(0,1), a(1,0), a(1,1), a(2,0), a(2,1)
- Fortran uses column-major order (and indexed from 1)
  - So, a(1,1), a(2,1), a(3,1), a(1,2), a(2,2), a(3,2)
  - What happens when you pass array references between Fortran and C code?
- Java does not have "real" 2-D arrays. A Java 2-D array is a pointer to a list of pointers to the rows

a(i,j) in C/C++/etc.

- To find a(i,j), we need to know
  - Values of i and j
  - How many *columns* the array has
- Location of a(i,j) is (assuming indexed from 0)

  Location of a + (i*(#of columns) + j)*element_size
- Can factor to pull out any load-time constant part and evaluate that at load time – no recalculating at runtime
  - E.g., a[1][5] becomes a + 15 if a has 10 columns and byte-sized elements.

Coming Attractions

- Code Generation for Objects
  - Representation
  - Method calls
  - Inheritance and overriding
- Strategies for implementing code generators
- Code improvement – optimization