ASM 1

Motivation

- Begin building an executable
- Also want to discuss more details of language semantics and specifying them

Tools

- We'll use NASM for x64
- Fetch NASM from `http://nasm.us`

Syntax

- ASM file needs to have layout like so:

```
default rel
section .text
...code...
.section .data
...globals...
```

Syntax

- We refer to locations in an ASM file with *labels*
- These are just names followed by a colon
  - Every named address is denoted with a label
  - No difference between a variable label, a function label, etc.
  - It's just an address

Example

- Suppose we had a C program like so:

```
int main(int argc, char* argv[]){
    return 0;
}
```

ASM

- This is the equivalent in assembly

```
default rel
section .text
global main
main:
    mov rax,0
    ret
section .data
```

Explanation

- The global directive tells nasm to make the symbol main visible outside the current file
- Then we define a label main
  - Tells where code for main() starts
- Put zero into register rax
- Then return

Registers

- Most of the time, values CPU works with go in registers
- There are sixteen of them that we normally use
  - rax, rcx, rdx, rbx, rsp, rbp, rsi, rdi, r8, r9, r10, r11, r12, r13, r14, r15
- Each one is 64 bits wide

Standard

- x64 ABI says that when a function returns, its return value is to be stored to rax
- That's why we move the value 0 to rax before returning
- Let's see another example...

Code

- Consider this C code...
- ```
  int x=42;
  int main(int argc, char* argv[]){
      return x;
  }
  ```
  ```

Assembly

```
default rel
section .text
global main
main:
    mov rax,[x]
    ret
section .data
x:
    dq 42
```

Explanation

- Global variable x
  - Define quadword of memory with initial value 42 (dq 42)
  - In main, pull value from memory location labeled x and put in rax
  - Then return

Example

- Another example:

```
int x=42;
int main(int argc, char* argv[]){
    return x+10;
}
```

Example

- Assembly:

```
default rel
section .text
global main
main:
    mov rax,[x]
    add rax,10
    ret
section .data
x:
    dq 42
```

Operations

- Basic math operations: Format: operation arg1 arg2
  - operation = MOV, ADD, SUB are most common
  - arg1 is both source and destination
    - Can be memory address or register
  - arg2 is second operand
    - Can be register or memory address or constant
    - Can't have two memory addresses in same instruction

Example

```
int x=4;
int y=5;
int z=0;
int main(int argc, char* argv[]){
    z = x+y;
    return z*3;
}
```

ASM

```
default rel
section .text
global main
main:
    mov rax,[x]
    add rax,[y]
    mov [z], rax        ;z=x+y
    mov rbx, 3
    imul rax, rbx       ;z*3
    ret
section .data
x:
    dq 4
y:
    dq 5
z:
    dq 0
```

Coding

- We'll get started with a simple grammar:
  program $\rightarrow$ braceblock
  stmts $\rightarrow$ stmt stmts | λ
  stmt $\rightarrow$ cond | loop | return-stmt SEMI
  loop $\rightarrow$ WHILE LP expr RP braceblock
  cond $\rightarrow$ IF LP expr RP braceblock | IF LP expr RP braceblock ELSE braceblock
  braceblock $\rightarrow$ LBR stmts RBR
  expr $\rightarrow$ NUM
  return-stmt $\rightarrow$ RETURN expr

Prerequisites

- Assume we've created the parse tree
- Assume our tree node looks something like this:

```
class TreeNode{
    public Token token; //might be null
    public string Symbol;
    public string Lexeme {
        get { return token.Lexeme }
    }
    public List<TreeNode> Children;
}
```

Prerequisites

- We'll create a function to use when we want to emit assembly code

```
static List<string> asmCode;
static void emit( string fmt, params object[] p){
    asmCode.Add( string.Format( fmt, p ) );
}
```

- By using string.Format, we can do interpolation of values
- Example:
  string src = "[x]"
  emit("mov rax, {0}" , src);

Bootstrap

```
void programNodeCode( TreeNode n ){
    //program -> stmts
    if( n.Symbol != "program" )
        ICE
    braceblockNodeCode( n.Children[0] );
}
```

- To begin, we call programNodeCode and pass it the root of the tree

braceblock

- This one is straightforward...

```
void braceblockNodeCode(TreeNode n){
    //braceblock -> LBR stmts RBR
    stmtsNodeCode(n.Children[1]);
}
```

stmts

- The next step is the stmts node processor:

```
void stmtsNodeCode(TreeNode n){
    //stmts -> stmt stmts | lambda
    if( n.Children.Count == 0 )
        return;
    stmtsNodeCode(n.Children[0]);
    stmtsNodeCode(n.Children[1]);
}
```

stmts

- Almost ready to do some real work...

```
void stmtNodeCode(TreeNode n){
    //stmt -> cond | loop | return-stmt SEMI
    var c = n.Children[0];
    switch( c.Symbol ){
        case "cond":
            condNodeCode(c); break;
        case "loop":
            loopNodeCode(c); break;
        case "return-stmt":
            returnstmtNodeCode(c); break;
        default: ICE
    }
}
```

Whew!

- All that was just preliminary setup
- Now we can look at how we handle cond, loop, and return

return

- Standard on x64: Return values go in rax

```
void returnstmtNodeCode(TreeNode n){
    //return-stmt -> RETURN expr
    exprNodeCode( n.Children[1] );
    ...move result from expr to rax...
    emit("ret");
}
```

- We need to write exprNodeCode now. Let's do that.

expr

```
void exprNodeCode(TreeNode n){
    //expr -> NUM
    What do?
}
```

expr

- We have to decide where to store the data
- Since we only have a single number, this is easy: We'll use one of the registers
- I'll use rax in these examples so we don't have to do any extra mov operations

expr

```
void exprNodeCode(TreeNode n){
    //expr -> NUM
    double d = Convert.ToDouble(n.Children[0].Lexeme);
    string ds = d.ToString("f");
    if(ds.IndexOf(".") == -1 )
        ds += ".0"; //nasm requirement
    emit("mov rax, __float64__({0})", ds);
}
```

Conditional Ops

- Pattern: We compute the result we want to test
- Then we compare that result to some constant (or variable)
- Then we jump over the 'then' block *if the condition is not satisfied*

Example

- We'll use some C code as an example:

```c
int x = 42;
int main(int argc, char* argv[]){
    if( x )
        return 1;
    return 0;
}
```

Code

- First, we have the conditional expression:
  x
  - In C: true if x is nonzero; false if x is zero
- So we fetch the value from x and store to a register
  mov rax, [x]

Code

- Next, we compare the value we fetched against 0:
  cmp rax, 0
- This sets some internal processor flags

Code

- Finally, we jump over the “return 1” code if x was false:
```
je notTrue
mov rax,1
ret
notTrue:
```
- Conditional jumps: jne (not equal), je (equal)

Else

- What if we have an else?
- We just extend the pattern: Need to jump over the else block when we're done with the then-block
- Example in C:

```
if( x ){
    y = 20;
    z = 40;
} else {
    w = 42;
}
```

Else

- We get:

```
    mov rax,[x]
    cmp x,0
    je elseBlock
    mov rax, 20
    mov [y], rax
    mov rax, 40
    mov [z], rax
    jmp endIf
elseBlock:
    mov rax, 42
    mov [w], rax
endIf:
    ...more code...
```

Our Language

- How do we apply this to our language?
- Let's take things one step at a time

Our Code

- Begin working on conditionals:

```
void condNodeCode(TreeNode n){
    //cond -> IF LP expr RP braceblock |
    //  IF LP expr RP braceblock ELSE braceblock
    exprNodeCode(n.Children[2]);
    emit("cmp rax,0");
    ...what now?...
}
```

label

- We'll need a function that can return a unique label for us to use...

```
static int labelCounter=0;
static string label(){
    string s = "lbl"+labelCounter;
    labelCounter++;
    return s;
}
```

cond

```
void condNodeCode(TreeNode n){
    //cond -> IF LP expr RP braceblock |
    //  IF LP expr RP braceblock ELSE braceblock
    exprNodeCode(n.Children[2]);
    emit("cmp rax,0");
    if( n.Children.Count == 5 ){
        var endifLabel = label();
        emit("je {0}",endifLabel);
        braceblockNodeCode(n.Children[4]);
        emit("{0}:", endifLabel);
    } else {
        ...fill this in...
    }
}
```

Example

- Suppose we have this input:

```
{
    if( 12 ){
        return 34;
    }
    return 56;
}
```

Assembly Code

```
default rel
section .text
global main
main:
    mov rax, __float64__(12.00)
    cmp rax,0
    je lbl0
    mov rax, __float64__(34.00)
    ret
lbl0:
    mov rax, __float64__(56.00)
    ret
section .data
```

Example 2

- Source code:

```
{
    if( 12 ){
        return 34;
    } else {
        return 78;
    }
    return 56;
}
```

Example 2

- Assembly output:

```
default rel
section .text
global main
main:
    mov rax, __float64__(12.00)
    cmp rax,0
    je lbl0
    mov rax, __float64__(34.00)
    ret
    jmp lbl1
lbl0:
    mov rax, __float64__(78.00)
    ret
lbl1:
    mov rax, __float64__(56.00)
    ret
section .data
```

Loops

- Loops are another place where we use conditional evaluation
- Example in C:

```
while( x ){
    ...
}
```

Code

```
loopStart:
    mov rax, [x]
    cmp rax, 0
    je loopEnd
    ...loop body statements...
    jmp loopStart
loopEnd:
```

Note

- An intelligent compiler might write the assembly like so:

```
    mov rax, [x]
    cmp rax, 0
    je loopEnd
loopStart:
    ...loop body statements...
    mov rax, [x]
    cmp x, 0
    jne loopStart
loopEnd:
```

- The code is a bit trickier to read, but it may execute faster (fewer jump operations)

Assignment

- Get the test harness working: Main.cs ExeTools.cs inputs.txt
  - You'll need to add the code for "loop" and finish the code for "cond"
- You can *not* assume the location or presence of any files on the disk. In particular, you won't have "grammar.txt" available.
  - You can embed the grammar as a C# string constant if you wish
  - Don't use resource streams: They won't be included in the executable that I build, and your program won't work as a result.
- One small problem remains...

Test Harness

- Automated test harnesses result in less work for the programmer: Easy to automatically test the code against a wide variety of inputs
- But: We don't have an easy way to print data or otherwise output it
- So: We rely on the fact that if a program returns to the OS, whatever is in rax is interpreted as its exit value
- But: Floating point values don't make valid exit values
- So: For the purposes of this test, we'll need to convert the value in rax from a floating point number to an integer value
- Alter the programNodeCode function a bit...

programNodeCode

```
void programNodeCode( TreeNode n ){
    //program -> stmts
    if(n.Symbol != "program")
        throw new Exception();
    emit("default rel");
    emit("section .text");
    emit("global main");
    emit("main:");
    emit("call theRealMain");
    emit("movq xmm0, rax");
    emit("cvtsd2si rax,xmm0");
    emit("ret");
    emit("theRealMain:");
    braceblockNodeCode( n.Children[0] );
    emit("ret");
    emit("section .data");
}
```

Sources

- Intel Corp. Intel Processor Reference Manuals.
- https://forum.nasm.us/index.php?topic=969.0

Created using LaTeX.

Main font: Gentium Book Basic, by Victor Gaultney. See http://software.sil.org/gentium/
Monospace font: Source Code Pro, by Paul D. Hunt. See https://fonts.google.com/specimen/Source+Code+Pro and http://sourceforge.net/adobe
Icons by Ulisse Perusin, Steven Garrity, Lapo Calamandrei, Ryan Collier, Rodney Dawes, Andreas Nilsson, Tuomas Kuosmanen, Garrett LeSage, and Jakub Steiner. See http://tango-project.org