PA5 Overview (for real this time)

Lecture 15

lol, let’s not target Windows executables

March 7, 2018
PA5 Overview

“[PA5] is by far the most difficult assignment... you have to get almost everything right which makes it difficult to estimate how much progress you’re making...”

“[PA5] was the most time intensive and the hardest of the assignments in both classes. There was (1) just a lot more stuff and (2) a lot more creativity required...”

“[PA5] needs a lot to come together to get a basic program working.”
Compiler Construction
Considerations for PA5

- Stack machine code generation
  - Without a sensible register allocator, we must make sure to use a fixed number of registers for everything.
- Vtables
- Object layout
- Interfacing with external libraries
x86-64

- CISC — Complex Instruction Set
- Lots of opportunities for optimizations
- Nightmarish interfacing with external libraries
- Must use external tools (e.g., gdb instead of the reference compiler)
- Amazing CV booster

Cool Assembly

- RISC — Reduced Instruction Set
- Only 8 registers
- Fewer optimization opportunities
- Reference compiler has helpful debugging capabilities
  - If you don’t like the reference compiler.... well, sorry
- Easier to complete

Compiler Construction
Cool Assembly

14  Mov of reg * reg
15  Call of reg
16  Call_L of label
17  Return
18  Jmp of label
19  Bz of reg * label
20  Bnz of reg * label
21  Blt of reg * reg * label
22  Ble of reg * reg * label
23  Beq of reg * reg * label
24  Push of reg
25  Pop of reg
26  Debug of reg
27  Trace
28  Add of reg * reg * reg
29  Sub of reg * reg * reg
30  Mul of reg * reg * reg
31  Div of reg * reg * reg
32  Li of reg * Int32.t
33  La of reg * label
34  Ld of reg * reg * Int32.t
35  St of reg * Int32.t * reg
36  Constant_Integer of Int32.t
37  Constant_Label of string
38  Constant_String of string
39  Comment of string
40  Alloc of reg * reg (* destination, size *)
41  Syscall of string
42  Preamble_Main
43  Raw_Text of string
44  AddI of reg * Int32.t
45  Strcmp of string * reg * reg * string
### Terminology and Formulas

- **Pointer to Raw Data**: Offset of section data within the executable file.
- **Size of Raw Data**: Amount of section data within the executable file.
- **RVA**: Relative Virtual Address. Memory offset from the beginning of the executable.
- **Virtual Address (VA)**: Absolute Memory Address (RVA + Base). The PE Header fields named VirtualAddress actually contain Relative Virtual Addresses.
- **Virtual Size**: Amount of section data in memory.
- **Base Address**: Offset in memory that the executable module is loaded.
- **ImageBase**: Base Address requested in the PE header of a module.
- **Module**: An PE formatted file loaded into memory. Typically EXE or DLL.
- **Pointer**: A memory address.
- **Entry Point**: The address of the first instruction to execute when the module is loaded.
- **Import**: DLl functions required for use by an executable module.
- **Export**: Functions provided by a DLL which may be imported by another module.
- **RVA/VA Conversion**: RVA = RVA + BaseAddress
- **RVA/VA Conversion**: VA = RVA + BaseAddress
- **RVA/VA Conversion**: VA = Raw Address
- **RVA/VA Conversion**: Raw = (RVA - SectionStartRVA) + (SectionStartRVA - SectionStartPtrToRaw)
- **RVA/VA Conversion**: VA = RVA + BaseAddress
- **RVA/VA Conversion**: VA = Raw - SectionStartPtrToRaw + (SectionStartRVA - ImageBase)
Stack Machines

- A simple evaluation model
- No variables or registers (aside from temporary storage)
- A stack of values for intermediate results
Example Stack Machine Program

- Consider two instructions
  - `push i` place integer $i$ on top of the stack
  - `add` pop two elements, add them and put the result back on the stack

- A program to compute $7 + 5$:
  ```
  push 7
  push 5
  add
  ```
Stack Machine Example

Each instruction:
- Takes its operands from the top of the stack
- Removes those operands from the stack
- Computes the required operation on them
- Pushes the result on the stack
Why Stack Machines?

- Each operation takes operands from the same place and puts the results in the same place (i.e., fixed offsets from the top of the stack)

- This means a uniform compilation scheme
  - To do an add, always get $sp[0]$ and $sp[1]$, add them, store result at $sp[0]$

- And thus a simpler compiler
  - This is the easiest way to do PA5
  - Register allocation is more complex!
Why Stack Machines?

- Location of the operands is implicit
  - Always on the top of the stack
- No need to specify operands explicitly
  - You can load fixed temporary registers with operands from stack!
  - example discipline:
    - load operand 1 by popping sp[0] to r4,
    - load operand 2 by popping sp[1] to r5
- No need to specify result location
  - e.g., always put result in r6, then push r6
- Can represent instruction as `add` instead of `add r1, r2`
  - Smaller program size (sometimes faster: why?)
  - Java Bytecode uses a stack evaluation model!
    - Dalvik uses register allocation
Remarks on Stack Machines

- The **add** instruction performs 3 memory operations
  - Two reads and one write to the stack

- How can we improve this?
  - Hint: **fold** from functional programming
 Remarks on Stack Machines

- The **add** instruction performs 3 memory operations
  - Two reads and one write to the stack

- How can we improve this?
  - Hint: **fold** from functional programming

- Idea: the top of the stack is accessed frequently
  - Keep an **accumulator** in a fixed register
  - Implement **add** as:
    - accumulator ← accumulator + top_of_stack
  - Only one memory operation!
Stack Machine with Accumulator

From before: $7 + 5$

```
acc ← 7
push 7
```

```
acc ← 5
```

```
acc ← acc + top
pop
```
Example: $3 + (7 + 5)$

<table>
<thead>
<tr>
<th>Code</th>
<th>Acc</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>acc ← 3</td>
<td>3</td>
<td>init</td>
</tr>
</tbody>
</table>
Example: $3 + (7 + 5)$

<table>
<thead>
<tr>
<th>Code</th>
<th>Acc</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>acc ← 3</td>
<td>3</td>
<td>init</td>
</tr>
<tr>
<td>push acc</td>
<td>3</td>
<td>init, 3</td>
</tr>
</tbody>
</table>
Example: $3 + (7 + 5)$

<table>
<thead>
<tr>
<th>Code</th>
<th>Acc</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>acc ← 3</td>
<td>3</td>
<td>init</td>
</tr>
<tr>
<td>push acc</td>
<td>3</td>
<td>init, 3</td>
</tr>
<tr>
<td>acc ← 7</td>
<td>7</td>
<td>init, 3</td>
</tr>
</tbody>
</table>
Example: $3 + (7 + 5)$

<table>
<thead>
<tr>
<th>Code</th>
<th>Acc</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>acc ← 3</td>
<td>3</td>
<td>init</td>
</tr>
<tr>
<td>push acc</td>
<td>3</td>
<td>init, 3</td>
</tr>
<tr>
<td>acc ← 7</td>
<td>7</td>
<td>init, 3</td>
</tr>
<tr>
<td>push acc</td>
<td>7</td>
<td>init, 3, 7</td>
</tr>
</tbody>
</table>
Example: $3 + (7 + 5)$

<table>
<thead>
<tr>
<th>Code</th>
<th>Acc</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>acc ← 3</td>
<td>3</td>
<td>init</td>
</tr>
<tr>
<td>push acc</td>
<td>3</td>
<td>init, 3</td>
</tr>
<tr>
<td>acc ← 7</td>
<td>7</td>
<td>init, 3</td>
</tr>
<tr>
<td>push acc</td>
<td>7</td>
<td>init, 3, 7</td>
</tr>
<tr>
<td>acc ← 5</td>
<td>5</td>
<td>init, 3, 7</td>
</tr>
</tbody>
</table>
Example: $3 + (7 + 5)$

<table>
<thead>
<tr>
<th>Code</th>
<th>Acc</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>acc $\leftarrow$ 3</td>
<td>3</td>
<td>init</td>
</tr>
<tr>
<td>push acc</td>
<td>3</td>
<td>init, 3</td>
</tr>
<tr>
<td>acc $\leftarrow$ 7</td>
<td>7</td>
<td>init, 3</td>
</tr>
<tr>
<td>push acc</td>
<td>7</td>
<td>init, 3, 7</td>
</tr>
<tr>
<td>acc $\leftarrow$ 5</td>
<td>5</td>
<td>init, 3, 7</td>
</tr>
<tr>
<td>acc $\leftarrow$ acc + top</td>
<td>12</td>
<td>init, 3, 7</td>
</tr>
</tbody>
</table>
**Example: 3 + (7 + 5)**

<table>
<thead>
<tr>
<th>Code</th>
<th>Acc</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>acc ← 3</td>
<td>3</td>
<td>init</td>
</tr>
<tr>
<td>push acc</td>
<td>3</td>
<td>init, 3</td>
</tr>
<tr>
<td>acc ← 7</td>
<td>7</td>
<td>init, 3</td>
</tr>
<tr>
<td>push acc</td>
<td>7</td>
<td>init, 3, 7</td>
</tr>
<tr>
<td>acc ← 5</td>
<td>5</td>
<td>init, 3, 7</td>
</tr>
<tr>
<td>acc ← acc + top</td>
<td>12</td>
<td>init, 3, 7</td>
</tr>
<tr>
<td>pop</td>
<td>12</td>
<td>init, 3</td>
</tr>
</tbody>
</table>
### Example: $3 + (7 + 5)$

<table>
<thead>
<tr>
<th>Code</th>
<th>Acc</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>acc ← 3</td>
<td>3</td>
<td>init</td>
</tr>
<tr>
<td>push acc</td>
<td>3</td>
<td>init, 3</td>
</tr>
<tr>
<td>acc ← 7</td>
<td>7</td>
<td>init, 3</td>
</tr>
<tr>
<td>push acc</td>
<td>7</td>
<td>init, 3, 7</td>
</tr>
<tr>
<td>acc ← 5</td>
<td>5</td>
<td>init, 3, 7</td>
</tr>
<tr>
<td>acc ← acc + top</td>
<td>12</td>
<td>init, 3, 7</td>
</tr>
<tr>
<td>pop</td>
<td>12</td>
<td>init, 3</td>
</tr>
<tr>
<td>acc ← acc + top</td>
<td>15</td>
<td>init, 3</td>
</tr>
</tbody>
</table>
**Example: 3 + (7 + 5)**

<table>
<thead>
<tr>
<th>Code</th>
<th>Acc</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>acc ← 3</td>
<td>3</td>
<td>init</td>
</tr>
<tr>
<td>push acc</td>
<td>3</td>
<td>init, 3</td>
</tr>
<tr>
<td>acc ← 7</td>
<td>7</td>
<td>init, 3</td>
</tr>
<tr>
<td>push acc</td>
<td>7</td>
<td>init, 3, 7</td>
</tr>
<tr>
<td>acc ← 5</td>
<td>5</td>
<td>init, 3, 7</td>
</tr>
<tr>
<td>acc ← acc + top</td>
<td>12</td>
<td>init, 3, 7</td>
</tr>
<tr>
<td>pop</td>
<td>12</td>
<td>init, 3</td>
</tr>
<tr>
<td>acc ← acc + top</td>
<td>15</td>
<td>init, 3</td>
</tr>
<tr>
<td>pop</td>
<td>15</td>
<td>init</td>
</tr>
</tbody>
</table>
Notes

- It is **critical** that the stack is preserved across the evaluation of a subexpression
  - Stack before evaluating \(7 + 5\) is \(init, 3\)
  - Stack after evaluating \(7 + 5\) is \(init, 3\)
  - The first operand is top of the stack
RISC-y Business

- We’ve been talking about some abstract notion of a stack machine

- The compiler actually generates assembly code for a specify ISA

- Briefly: COOL-ASM is a RISC-style assembly language
  - Untyped, unsafe, low-level, few primitives
  - 8 general registers, 3 special registers sp, fp, and ra
  - load-store architecture: bring values into registers from memory to operate on them
Drink your Cool-aid

Sample COOL-ASM instructions
(more in CRM)

1. add r2 <- r5 r2 ; r2 = r5 + r2
2. li r5 <- 183 ; r5 = 183
3. ld r2 <- r1[5] ; r2 = *(r1 + 5)
4. st r1[6] <- r7 ; *(r1 + 6) = r7
5. my_label:
6. push r1 ; *sp = r1; sp--;
7. sub r1 <- r1 1 ; r1 = r1 - 1
8. bnz r1 my_label ; if (r1 != 0) goto my_label
Simulating a Stack Machine

- The compiler generates COOL-ASM (or x86-64)
  - Emit code to implement a stack machine!
- Convention: accumulator is kept in register r1
- Stack is stored in memory
  - We have more memory than registers
- Both x86-64 and COOL-ASM have supply stacks that grow downwards
- The address of the next unused location on the stack is kept in register sp
  - i.e., the top of the stack is at address sp+1
  - ld r1 <- sp[1] ; load top of stack into acc
- Word size = 1 in COOL-ASM (8 in x86-64)
COOL-ASM for 7+5

Stack Machine
acc <- 7
push acc
acc <- 5
acc <- acc + top
pop

COOL-ASM
li r1 7
push r1
li r1 5
pop r2
add r1 <- r1 r2
pop r2
Generalizing Code Generation

- Cool is an expression-based language...
  - We want to take a principled approach that lets us generate code for every type of expression, no matter where the expression appears in the code, and regardless of how complex the expressions are

- We make a `cgen(e)` function that generates code for an expression in the AAST

- Follow a discipline: `cgen(e)` should compute the value of `e` and store it in `r1` (the accumulator)

- Preserve `sp` and the contents of the stack
  - No matter what the expression must do to the stack, leave it the way you found it
Code Generation: Constants

cgen(123) = li r1 123

Just move the constant into the accumulator! This also preserves the stack (doesn’t touch it)
Code generation: Addition

cgen(e_1 + e_2) =
cgen(e_1)
push r1
cgen(e_2);
;; e2 now in r1
pop r2
;; r2 contains the saved result of evaluating r1!
add r1 <- r1 r2

Why can’t we use r2 directly after cgen(e1) to save a push and pop?
Code generation notes

- The cgen code for $+$ is a template with holes for code that evaluates $e_1$ and $e_2$

- Stack Machine code generation is recursive

- Code for $e_1 + e_2$ consists of code for $e_1$ and $e_2$ glued together

- Most critically! Code generation can be written as a recursive-descent tree walk of the AAST
  - At least, for expressions
Code Generation: If

cgen(if e1 = e2 then e3 else e4 fi) =
  cgen(e1)
push r1
cgen(e2)
pop r2
beq r1 r2 true_branch ;; else fall through
bne r1 r2 else_branch
true_branch:
cgen(e3)
jmp after_if
else_branch:
cgen(e4)
jmp after_if
after_if:
Code Generation: Variables

► “Variables” could be function parameters, let bindings...
  ► Parameters get pushed on the stack
  ► Recall: fp is established by the function prologue, sp can change during function execution

Consider: \( f(x_1, x_2) \)

\[
\begin{align*}
x_1 \text{ is at } & \text{fp + 2} \\
x_2 \text{ is at } & \text{fp + 1}
\end{align*}
\]

Thus:
\[
cgen(x_i) = \text{ld r1 <- fp[z]} \\
(z \approx n+1 - i)
\]
Summary

- Activation record is co-designed with the code generator

- Code generation is a recursive traversal of the AAST

- Stack Machine code is simpler for PA5 (although you’re encouraged to try register allocation!)
Object Oriented Code Generation

- “Variables” are references to objects
  - What are the semantics of addition in Cool?

- Liskov Substitution Principle: If B is a subclass of A, then an object of class B can be used wherever an object of class A is expected

- This means that code in class A must work unmodified on an object of class B
Two issues

- How are objects represented in memory?
- How is dynamic dispatch implemented?
Object Layout

- An object is like a `struct` in C. The reference `foo.field` is an index into a `foo` struct at an offset corresponding to `field`:
  - Objects are laid out in contiguous memory
  - Each attribute stored at a fixed offset in object
  - When a method is invoked, the object becomes `self` and the fields are the object’s attributes
Cool Object Layout

- **Class tag**
  - An integer that identifies the class of the object (Int=1, Bool=2, ...)

- **Object size**
  - Number of words occupied by the object

- **Dispatch pointer**
  - Address to a table of methods

- **Attributes**
  - Each attribute laid out in contiguous memory