Course staff

Will

Stephanie

Esther

Varun
Today’s goals

• Course overview

• Intro to syntax and semantics
Course overview
Problem #1:
What is a programming language?
“A programming language is a notation for writing programs, which are specifications of a computation or algorithm.”
- Wikipedia

“A vocabulary and set of grammatical rules for instructing a computer to perform specific tasks.”
- Fundamental of Programming Languages (Ellis Horowitz)

“Programming languages are the medium of expression in the art of computer programming.”
- Concepts in Programming Languages (John Mitchell)

“A good programming language is a conceptual universe for thinking about programming”.  
- Alan Perlis
When in doubt: majority vote!

http://etc.ch/PvnC
Problem #2:
How do we describe programming languages?
## CVTTPS2DQ—Convert with Truncation Packed Single-Precision Floating-Point Values to Packed Signed Doubleword Integer Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 OF 5B /r CVTTPS2DQ xmm1, xmm2/m128</td>
<td>RM</td>
<td>V/V</td>
<td>SSE2</td>
<td>Convert four packed single-precision floating-point values from xmm2/mem to four packed signed doubleword values in xmm1 using truncation.</td>
</tr>
<tr>
<td>VEX.128.F3.0F.WIG 5B /r CVTTPS2DQ xmm1, xmm2/m128</td>
<td>RM</td>
<td>V/V</td>
<td>AVX</td>
<td>Convert four packed single-precision floating-point values from xmm2/mem to four packed signed doubleword values in xmm1 using truncation.</td>
</tr>
<tr>
<td>VEX.256.F3.0F.WIG 5B /r CVTTPS2DQ ymm1, ymm2/m256</td>
<td>RM</td>
<td>V/V</td>
<td>AVX</td>
<td>Convert eight packed single-precision floating-point values from ymm2/mem to eight packed signed doubleword values in ymm1 using truncation.</td>
</tr>
<tr>
<td>EVEX.128.F3.0F.W0 5B /r CVTTPS2DQ xmm1 {k1}[z], xmm2/m128/m32bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed signed doubleword values in xmm1 using truncation subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.F3.0F.W0 5B /r CVTTPS2DQ ymm1 {k1}[z], ymm2/m256/m32bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed signed doubleword values in ymm1 using truncation subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.F3.0F.W0 5B /r CVTTPS2DQ zmm1 {k1}[z], zmm2/m512/m32bcst {sae}</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Convert sixteen packed single-precision floating-point values from zmm2/m512/m32bcst to sixteen packed signed doubleword values in zmm1 using truncation subject to writemask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>FV</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Converts four, eight or sixteen packed single-precision floating-point values in the source operand to four, eight or sixteen packed signed doubleword values in the destination operand using truncation.
6.7.3.1 Formal definition of restrict

1. Let $D$ be a declaration of an ordinary identifier that provides a means of designating an object $P$ as a restrict-qualified pointer to type $T$.

2. If $D$ appears inside a block and does not have storage class `extern`, let $B$ denote the block. If $D$ appears in the list of parameter declarations of a function definition, let $B$ denote the associated block. Otherwise, let $B$ denote the block of `main` (or the block of whatever function is called at program startup in a freestanding environment).

3. In what follows, a pointer expression $E$ is said to be based on object $P$ if (at some sequence point in the execution of $B$ prior to the evaluation of $E$) modifying $P$ to point to a copy of the array object into which it formerly pointed would change the value of $E$.\(^{119}\) Note that “based” is defined only for expressions with pointer types.

4. During each execution of $B$, let $L$ be any lvalue that has $&L$ based on $P$. If $L$ is used to access the value of the object $X$ that it designates, and $X$ is also modified (by any means), then the following requirements apply: $T$ shall not be const-qualified. Every other lvalue used to access the value of $X$ shall also have its address based on $P$. Every access that modifies $X$ shall be considered also to modify $P$, for the purposes of this subclause. If $P$ is assigned the value of a pointer expression $E$ that is based on another restricted pointer object $P_2$, associated with block $B_2$, then either the execution of $B_2$ shall begin before the execution of $B$, or the execution of $B_2$ shall end prior to the assignment. If these requirements are not met, then the behavior is undefined.

5. Here an execution of $B$ means that portion of the execution of the program that would correspond to the lifetime of an object with scalar type and automatic storage duration

\(^{119}\) In other words, $E$ depends on the value of $P$ itself rather than on the value of an object referenced by $P$. This allows for situations where $P$ is modified before $E$ is evaluated, changing the value of $E$. The term “based” emphasizes that this dependency is specific to pointers and not other types of expressions.
<table>
<thead>
<tr>
<th>Language</th>
<th>files</th>
<th>blank</th>
<th>comment</th>
<th>code</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>32429</td>
<td>556567</td>
<td>579624</td>
<td>2929550</td>
</tr>
<tr>
<td>Ada</td>
<td>5711</td>
<td>277381</td>
<td>368655</td>
<td>783143</td>
</tr>
<tr>
<td>C++</td>
<td>22434</td>
<td>169922</td>
<td>226650</td>
<td>747203</td>
</tr>
<tr>
<td>P0 File</td>
<td>42</td>
<td>269374</td>
<td>363807</td>
<td>715169</td>
</tr>
<tr>
<td>Go</td>
<td>3665</td>
<td>72849</td>
<td>108674</td>
<td>542915</td>
</tr>
<tr>
<td>C/C++ Header</td>
<td>3183</td>
<td>123298</td>
<td>162805</td>
<td>522281</td>
</tr>
<tr>
<td>Markdown</td>
<td>437</td>
<td>50424</td>
<td>0</td>
<td>414990</td>
</tr>
<tr>
<td>Bourne Shell</td>
<td>142</td>
<td>77080</td>
<td>63019</td>
<td>400052</td>
</tr>
<tr>
<td>Fortran 90</td>
<td>5376</td>
<td>25125</td>
<td>41675</td>
<td>142157</td>
</tr>
<tr>
<td>#1</td>
<td>198</td>
<td>8110</td>
<td>2549</td>
<td>71077</td>
</tr>
<tr>
<td>Assembly</td>
<td>585</td>
<td>12706</td>
<td>31134</td>
<td>62206</td>
</tr>
<tr>
<td>XML</td>
<td>61</td>
<td>6307</td>
<td>543</td>
<td>46731</td>
</tr>
<tr>
<td>Windows Module Definition</td>
<td>150</td>
<td>6171</td>
<td>35</td>
<td>46472</td>
</tr>
<tr>
<td>Expect</td>
<td>326</td>
<td>6786</td>
<td>12325</td>
<td>28625</td>
</tr>
<tr>
<td>HTML</td>
<td>114</td>
<td>523</td>
<td>29</td>
<td>26936</td>
</tr>
<tr>
<td>Objective C</td>
<td>525</td>
<td>4940</td>
<td>3162</td>
<td>16966</td>
</tr>
<tr>
<td>Perl</td>
<td>23</td>
<td>1470</td>
<td>2234</td>
<td>14135</td>
</tr>
<tr>
<td>Make</td>
<td>122</td>
<td>2222</td>
<td>1464</td>
<td>13327</td>
</tr>
<tr>
<td>Fortran 77</td>
<td>502</td>
<td>1415</td>
<td>4326</td>
<td>11920</td>
</tr>
<tr>
<td>Objective C++</td>
<td>247</td>
<td>2487</td>
<td>1567</td>
<td>8178</td>
</tr>
<tr>
<td>TeX</td>
<td>4</td>
<td>826</td>
<td>3308</td>
<td>6557</td>
</tr>
<tr>
<td>Python</td>
<td>27</td>
<td>1669</td>
<td>1815</td>
<td>6199</td>
</tr>
<tr>
<td>Pascal</td>
<td>24</td>
<td>1335</td>
<td>6949</td>
<td>5321</td>
</tr>
<tr>
<td>MSBuild script</td>
<td>7</td>
<td>1</td>
<td>0</td>
<td>4675</td>
</tr>
<tr>
<td>awk</td>
<td>21</td>
<td>548</td>
<td>792</td>
<td>3877</td>
</tr>
<tr>
<td>Fortran 95</td>
<td>98</td>
<td>898</td>
<td>2412</td>
<td>2680</td>
</tr>
<tr>
<td>Bourne Again Shell</td>
<td>20</td>
<td>457</td>
<td>689</td>
<td>2034</td>
</tr>
<tr>
<td>C#</td>
<td>9</td>
<td>230</td>
<td>506</td>
<td>879</td>
</tr>
<tr>
<td>JSON</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>384</td>
</tr>
<tr>
<td>yacc</td>
<td>1</td>
<td>56</td>
<td>37</td>
<td>316</td>
</tr>
<tr>
<td>OCaml</td>
<td>1</td>
<td>44</td>
<td>29</td>
<td>285</td>
</tr>
<tr>
<td>Standard ML</td>
<td>1</td>
<td>34</td>
<td>28</td>
<td>215</td>
</tr>
<tr>
<td>vim script</td>
<td>4</td>
<td>50</td>
<td>71</td>
<td>193</td>
</tr>
<tr>
<td>CMake</td>
<td>1</td>
<td>32</td>
<td>31</td>
<td>186</td>
</tr>
<tr>
<td>lex</td>
<td>1</td>
<td>34</td>
<td>30</td>
<td>154</td>
</tr>
<tr>
<td>Haskell</td>
<td>38</td>
<td>17</td>
<td>0</td>
<td>122</td>
</tr>
<tr>
<td>NAnt script</td>
<td>2</td>
<td>17</td>
<td>0</td>
<td>103</td>
</tr>
<tr>
<td>Windows Resource File</td>
<td>3</td>
<td>5</td>
<td>3</td>
<td>96</td>
</tr>
<tr>
<td>MATLAB</td>
<td>3</td>
<td>13</td>
<td>0</td>
<td>46</td>
</tr>
<tr>
<td>SAS</td>
<td>1</td>
<td>14</td>
<td>22</td>
<td>32</td>
</tr>
<tr>
<td>Brainfuck</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>SWC</td>
<td>1</td>
<td>2</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>Lisp</td>
<td>1</td>
<td>4</td>
<td>12</td>
<td>8</td>
</tr>
<tr>
<td>DOS Batch</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>CSS</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**SUM:**                         | 76532 | 1680566 | 1991214   | 7587239  |
move_if_noexcept obtains an rvalue reference to its argument if its move constructor does not throw exceptions or if there is no copy constructor (move-only type), otherwise obtains an lvalue reference to its argument. It is typically used to combine move semantics with strong exception guarantee.

**Parameters**

- `x` - the object to be moved or copied

**Return value**

`std::move(x)` or `x`, depending on exception guarantees.

**Notes**

This is used, for example, by `std::vector::resize`, which may have to allocate new storage and then move or copy elements from old storage to new storage. If an exception occurs during this operation, `std::vector::resize` undoes everything it did to this point, which is only possible if `std::move_if_noexcept` was used to decide whether to use move construction or copy construction. (unless copy constructor is not available, in which case move constructor is used either way and the strong exception guarantee may be waived)

**Example**
Naming and Binding

Names refer to objects. Names are introduced by name binding operations. Each occurrence of a name in the program text refers to the binding of that name established in the scope containing the use.

A code block is a piece of Python program text that is executed as a unit. The following are blocks: a module, a function body, and a class definition. Each command typed interactively in a file given as standard input to the interpreter or specified on the interpreter command line the first argument is a code block. A script command (a command specified on the command line with the ‘-c’ option) is a code block. The file read by the built-in function `execfile()` is a code block. The string argument passed to the built-in function `eval()` and the code block. The expression read and evaluated by the built-in function `input()` is a code block.

A code block is executed in an execution frame. A frame contains some administrative information (used for debugging) and determines where and how execution continues when the execution has completed.

Scope defines the visibility of a name within a block. If a local variable is defined in a block, its scope includes that block. If the definition occurs in a function block, the scope is the same as the block in which it is defined within the defining one, unless a contained block introduces a different binding for the name. The scope of names defined in a class block is limited to the class block and any code blocks of methods – this includes generator expressions since they are implemented using a function scope. This means that the following will fail:

```python
A:
```
```
a = 42
b = list(a + i for i in range(10))
```

If a name is used in a code block, it is resolved using the nearest enclosing scope. The set of all such scopes visible to a code block is called the block’s environment.

A name is bound in a block, it is a local variable of that block. If a name is bound at the module level, it is a global variable. (The variables of the module code block are local and bound in that code block but not defined there, it is a free variable.

If a name is not found at all, a `NameError` exception is raised. If the name refers to a local variable that has not been bound, a `UnboundLocalError` exception is raised. They are class of `NameError`.

Following constructs bind names: formal parameters to functions, `import` statements, class and function definitions (these bind the class or function name in the defining block). Names are bound in an assignment, for loop header, in the second position of an except clause header or after as in a with statement. The `import` statement of the form `as name` names defined in the imported module, except those beginning with an underscore. This form may only be used at the module level.

A `get` occurring in a `del` statement is also considered bound for this purpose (though the actual semantics are to unbind the name). It is illegal to unbind a name that is referenced elsewhere; the compiler will report a `SyntaxError`.

A named assignment or import statement occurs within a block defined by a class or function definition or at the module level (the top-level code block).

A name binding operation occurs anywhere within a code block, all uses of the name within the block are treated as references to the current block. This can lead to errors if occurring in a block before it is bound. This rule is subtle. Python lacks declarations and allows name binding operations to occur anywhere within a code block. The local variables are determined by scanning the entire text of the block for name binding operations.

A global statement occurs within a block, all uses of the name specified in the statement refer to the binding of that name in the top-level namespace. Names are resolved by searching the global namespace, i.e. the namespace of the module containing the code block, and the builtins namespace, the namespace of the module the execution is searched first. If the name is not found there, the builtins namespace is searched. The global statement must precede all uses of the name.

The builtins namespace associated with the execution of a code block is actually found by looking up the name `__builtins__` in its global namespace; this should be a dictionary (otherwise the module’s dictionary is used). By default, when in the `main` module, `__builtins__` is the built-in module `builtins` (note: no ‘c’); when in any other module, `__builtins__` is the built-in module `sys`. When in a namespace, the builtins provided by the namespace module are used.
Problem #3:
How do we reason about programming languages?
Annual cost of software bugs

- $59 billion
- $1.1 trillion

Sources: NIST 2002, Tricentis 2017
Annual cost of software bugs (extrapolated)

Sources: NIST 2002, Tricentis 2017
Type safety matters more than ever
Dennard scaling is dead
High-performance DSLs reign

```python
a = tf.placeholder(tf.int16)
b = tf.placeholder(tf.int16)

# Define some operations
add = tf.add(a, b)
mul = tf.multiply(a, b)

# Launch the default graph.
with tf.Session() as sess:
    # Run every operation with variable input
    print("Addition with variables: %i" % sess.run(add, feed_dict={a: 2, b: 3}))
    print("Multiplication with variables: %i" % sess.run(mul, feed_dict={a: 2, b: 3}))
```

TensorFlow

```
Func blur_3x3(Func input) {
    Func blur_x, blur_y;
    Var x, y, xi, yi;

    // The algorithm - no storage or order
    blur_x(x, y) = (input(x-1, y) + input(x, y) + input(x+1, y))/3;
    blur_y(x, y) = (blur_x(x, y-1) + blur_x(x, y) + blur_x(x, y+1))/3;

    // The schedule - defines order, locality; implies storage
    blur_y.tile(x, y, xi, yi, 256, 32)
        .vectorize(xi, 8).parallel(y);
    blur_x.compute_at(blur_y, x).vectorize(x, 8);

    return blur_y;
}
```

Halide
Course theme:
Bridging the gap between systems and theory
Course overview

• **Theory**
  - Formal systems for describing and reasoning about PLs
  - Core vocabulary for describing programming constructs
  - Essentials of the functional programming paradigm

• **Systems**
  - Apply formal methods to real world languages and systems
  - Reason about memory safety, state machines, assembly, …
  - Compare value of dynamic vs. static typing
Syllabus

- **Weeks 1-3: Theory**
  - Lambda calculus and OCaml
  - The language of programming languages
  - Functional programming basics

- **Weeks 4-5: WebAssembly**
  - Case study on applying formal semantics to low level languages

- **Weeks 5-7: Rust**
  - Memory safety, traits, concurrency, state machines, and communication
  - (Plus a WebAssembly interpreter!)

- **Weeks 8-9: Lua**
  - Dynamic vs. static typing, object systems
Course structure

• Weekly assignments
  - Submit up to 3 days late per assignment, 5 late days total over semester
  - Mixed programming/written

• No midterm
  - … But there is a final

• No required readings
  - Supplementary material in the syllabus
  - All lecture notes/code will be posted online
  - Lecture videos available through SCPD
Prerequisites

- **CS 103**: induction, first order logic, basic proofs

- **CS 110**: assembly (ARM or x86), C, concurrency (threads, synchronization primitives)
Intro to syntax and semantics