# Review for Midterm

Prof. Stephen A. Edwards



## What Have We Covered?

- General Language Issues
- Assembly Languages
- C
- C++



## General Language Issues

- Syntax, Semantics, and Models of Computation
- Specification versus Modeling
- Concurrency: Two things at once
- Nondeterminsm: Unpredictability
- Types of communication: Memory, broadcasting
- Hierarchy



## Models of Computation

- All languages we have studied thus far use the same model of computation:
  - Imperative program operating on a memory space

Fetch an instruction

Read its operands

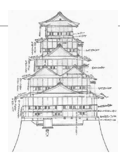
Perform the action

Save the results

Go on to the next instruction



## Specification and Modeling



- How do you want to use the program?
- Specification languages say “build this, please”
- Modeling languages allow you to describe something that does or will exist
- Distinction a function of the model and the language’s semantics

Copernican Model of the Solar System



## Nondeterminism

- You simply cannot predict what will happen
- No statistical distribution, no expected behavior
- It may not work, work for the moment and fail, or always work
- You saw this in the homework assignment
- Nondeterministic language allows nondeterministic programs

## Assembly Languages

- Program a sequence of instructions
- Embodies the Von Neumann model of computation:
- fetch, read, execute, store

- Instructions consist of opcode and operands
- Registers and addressing modes



## CISC Assembly Language

- Designed for humans to write

- Often fewer, special-purpose registers
- Single instruction can perform a lot of work
- Two-address instructions (source1, source2/dest)
- Difficult to pipeline
- Difficult compiler target (hard to model)



## RISC Assembly Language

- Simple, more orthogonal
- Three-operand instructions (source1, source2, dest)
- More, uniformly-accessible registers
- Many have delayed branch instructions

```
j MyLabel
add R1, R2, R3    % Executed after the jump instruction
sub R2, R3, R4    % Not executed
```

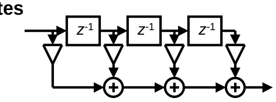


## Main DSP Application

- Finite Impulse Response filter (FIR)
- Can be used for lowpass, highpass, bandpass, etc.
- Basic DSP operation

- For each sample, computes

$$y_n = \sum_{i=0}^{k} a_i \, x_{n+i}$$




- $a_0 \ldots a_k$ are filter coefficients
- $x_n$ and $y_n$ are the nth input and output sample



## Traditional DSP Architectures

- Multiply-accumulate operation central
- Small number of special-purpose registers
- Stripped-down datapath to maximize speed, minimize cost, power
- Difficult to program automatically
- Specialized instruction-level parallelism
- Architecture heavily specialized to application domain
  - Complex addressing modes
  - MAC instruction
  - Limited zero-overhead loops



## VLIW Architectures

- Next step on the path toward more instruction-level parallelism
- More orthogonal: more costly, but more flexible than traditional DSPs
- Bigger register banks
- Simple RISC-like instructions issued in parallel
- Multiple, slightly differentiated computational units

- Virtually impossible to program by hand
- Reasonable compiler target

## The C Language

- High-level assembly for systems programming
- Originally used to develop the Unix operating system
- Pragmatic language as a result
- Stack-frame based mechanism for recursion, automatic variables
- Low-level model of memory inherited from typeless BCPL
- Influenced its view of arrays, pointers



## C Programs

- Collection of Functions
  - Recursive
  - Automatic (local) variables
- Functions contain statements
  - Simple control-flow (if-else, for, while, switch)
- Statements contain expressions
  - Powerful menagerie of operators
  - Arithmetic, logical, bit-oriented, comparison, assignment



## C Types

- Based on processor's natural types
- (Actually, a PDP-11's natural types)
- Integers
- Floating-point numbers
- Bytes (characters)
- Funny declarator syntax
  - int (*f)(double, int)



## C Structs and Unions

- Struct:
- Way to group objects in memory
- Padded to guarantee alignment requirements
- Each field given its own storage
- Union:
- Way to store different objects in the same space
- Size equal to size of largest element
- Each field stored in the same place



## Dynamic Memory Management

- Malloc() and free() system calls
- Maintains a "free list" of available storage
- Malloc() locates suitable storage, or requests more from OS if necessary
- Free() release its given area to free list, updates the data structure
- Can be slow and unpredictable
- Time/space overhead



## C Arrays

- View left over from BCPL's typeless view of memory
- a[k] is equivalent to a + k (pointer arithmetic)
- Thus a[0] is the base of the array
- Objects in array simply tiled

## C Operators

- **Arithmetic + ***
- **Logical & |**
- **Lazy logical && || (expand to conditional branches)**
- **Pointer arithmetic allowed (from BCPL)**



## setjmp/longjmp

- **A way to exit from deeply nested functions**

```
#include <setjmp.h>          Stores a jump target
jmp_buf jmpbuf;

setjmp(jmpbuf);              Stores context, returns 0


longjmp(jmpbuf,k);           Jumps back to target in
                             jmpbuf
```



## Setjmp/longjmp

- **The weird part: longjmp sends control back to the setjmp call that initialized the jmp_buf**

```
switch (setjmp(jmpbuf)) {
  case 0: /* first time */ break;
  case 1: /* longjmp called */ break;
}
```

- **It's as if setjmp returns twice**



## Using setjmp/longjmp

```
#include <setjmp.h>
jmp_buf jmpbuf;

int main(int argc, char *argv[]) {

  switch (setjmp(jmpbuf)) {
  case 0:
    body();  /* Normal program execution */
    break;
  case 1:
    error("something bad!");
    break;
  }
}
```



## Using setjmp/longjmp

- **Where an error occurs**

```
if ( having_trouble )
  longjmp(jmpbuf, ERROR_CODE);
```

- **Will exit this function as well as others currently being executed**
- **Does not do any clean-up on the way**



## C++

- **C with facilities for structuring very large programs**
- **Classes for new data types**
- **Operator overloading for convenient arithmetic expressions**
- **References for pass-by-name arguments**
- **Inline functions for speed**
- **Templates for polymorphism**
- **Exceptions**
- **Vast standard library**

## Classes

- Extension of C struct that binds functions to the object
- Inheritance: adding new fields, methods to an existing class to build a new one
- Object layout model
  - Single inheritance uses a trick
  - New data members simply tacked on at the end
  - Can't remove data members in derived classes
  - Multiple inheritance more complicated



## Virtual Functions

- Normal methods dispatched by the static type of the object determined at compile time
- Virtual functions dispatched by the actual type of the object at run time

```
struct A {                struct B : A {
  void f();                 void f();
  virtual void g();         virtual void g();
};                        };

A* a = new B;
a->f();       // calls A::f()
a->g();       // calls B::g()
```



## Implementing Virtual Functions

- Each object of a class with virtual functions has an extra pointer to its virtual table
- Virtual table has pointers to the virtual functions for the class
- Compiler fills in these virtual tables



## Const

- Way to pass pointers to objects that should not be modified

```
void g(char *a, const char *b);
void f(char *a, const char *b) {
  *a = 'a';        // OK
  *b = 'b';        // Error: b is const
  g(a,a);          // OK: non-const cast to const
  g(b,b);          // Error: const b cast to non-const
}
```



## Inline

- C++ can "inline" function calls: copy the function's body to the call site

```
inline int sum(int a, int b) { return a + b; }

c = sum(5, 6);
```

is compiled as

```
c = 5 + 6;
```



## FAQs

- Do we need to know each assembly language in detail for the test?

*No: I want you to understand the structure of the assembly languages.*

- Will the test require writing a big program?

*Not a big one, but perhaps a small one.*

- Are C++ compilers implemented in one pass like C compilers?

*Definitely not. C++ is much too complex. Modern C compilers make multiple passes, too.*

## Program Size Versus Speed

- Not always a direct trade-off

- Dumb example:

```
int sum(int a, int b) {
  return a + b;
}

c = sum(5,6) + sum(7,8);
```

```
int sum1(int a, int b) {
  return a + b;
}

int sum2(int a, int b) {
  return a + b;
}

c = sum1(5,6) + sum2(7,8);
```



## Maybe not so dumb

```
Template <class T> sort(int size, T* array) { … }

char *c[10];
sort<char *>(10,c);
float *c[10];
sort<float *>(10,c);
```

- Each call of sort will generate a distinct, identical copy of the code for sort