18-344 Recitation 2

Learning the Tool Chain

Logistics

- HW 1 released via gradescope due Sept 14, BEFORE CLASS
  - Correction: Q2The **latency** of specific instructions
- Lab0 was due Sept 5.
- Lab1 releases Sept 14, Due Sept 21.

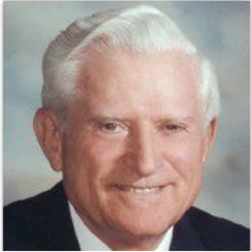
Recitation 1 Corrections

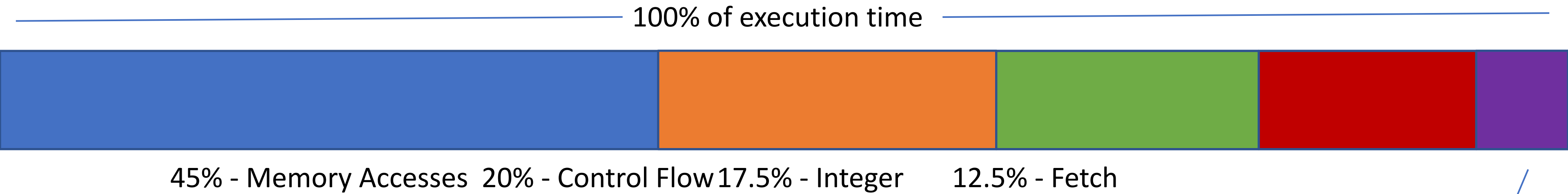
Spec2017 Failing Benches:

- deepsjeng, perlbench, specrandom
- x264
- No fails

Review

Amdahl's Law





5% - Floating Point

Amdahl's Law:

optimized time = [1-p x time / 1.0] + [p x time / speedup]

Or equivalently:

speedup = 1 / [(1 – p) / 1.0 + p / speedup]

Another view of the world: Gustafson’s Law

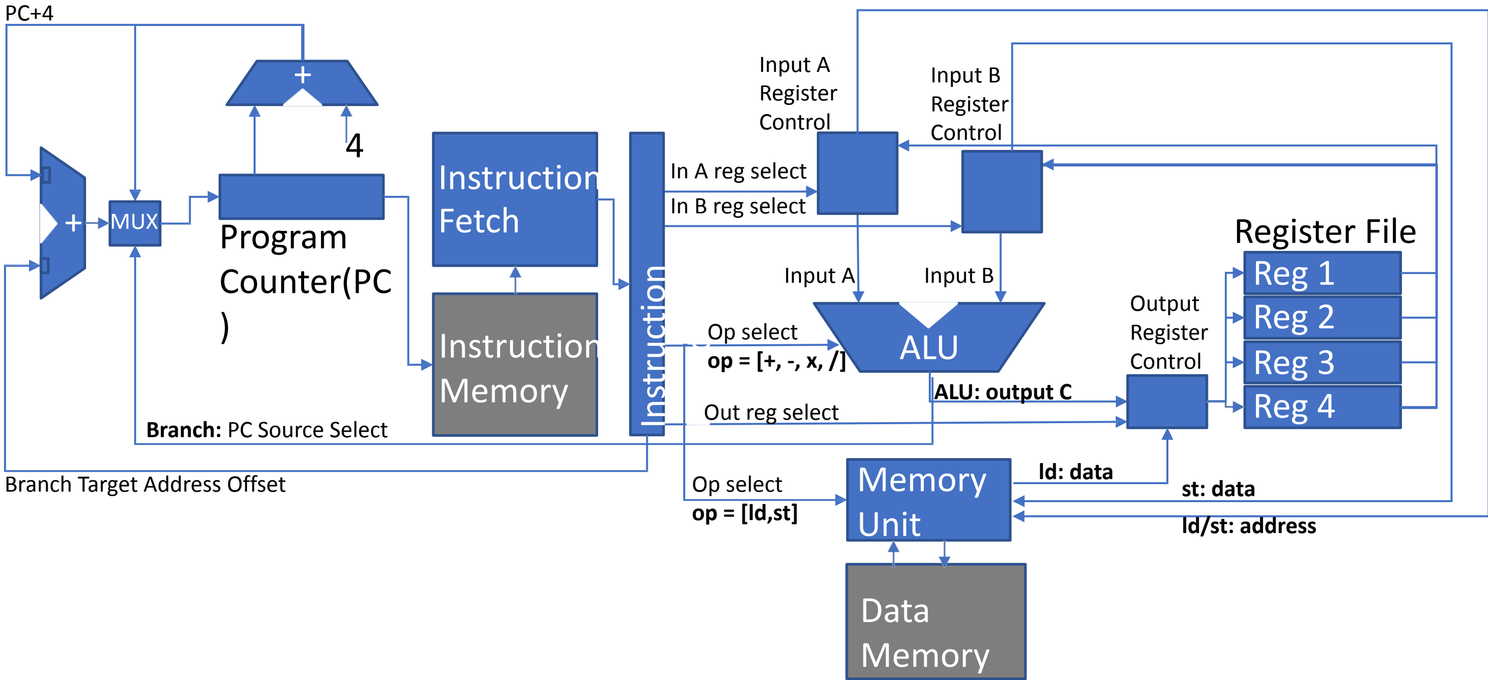
Gustafson’s Law for overall speedup with speedup factor of N:

(assume) Optimized time = $T = 1$

Unoptimized time = $T' = (1-p)T + pT*N = (1-p) + pN$

Scaled Speedup = $T' / T = (1-p) + pN$

A Complete (but slightly messy) RISCV-ish Datapath



What should go in the ISA?

Reduced Instruction Set Computer

Simple primitives:
Let software compose complex operations

Register operands:
Decouple functionality from memory accesses

Few total operations:
Usually only one way to do something

Few cases to map to control signals in microarchitecture

```
rd = M[imm]
rd = M[reg]
rd = M[reg + imm]
rd = M[PC + imm]
```

Complex Instruction Set Computer

Simple & complex operations:
Hardware must support complex functionality

Many operations:
Often several ways to do the same thing

Register and memory operands:
Operations may directly manipulate memory

Many cases to map to control signals in microarchitecture

	Source	Dest	Src,Dest	C Analog
movq	*Imm*	*Reg*	`movq $0x4,%rax`	`temp = 0x4;`
		Mem	`movq $-147,(%rax)`	`*p = -147;`
	Reg	*Reg*	`movq %rax,%rdx`	`temp2 = temp1;`
		Mem	`movq %rax,(%rdx)`	`*p = temp;`
	Mem	*Reg*	`movq (%rax),%rdx`	`temp = *p;`

Remember this from 18-213?

Plus all of these combinations

D(Rb,Ri,S) **Mem[Reg[Rb]+S*Reg[Ri]+ D]**

Principles of ISA Design

General Principles

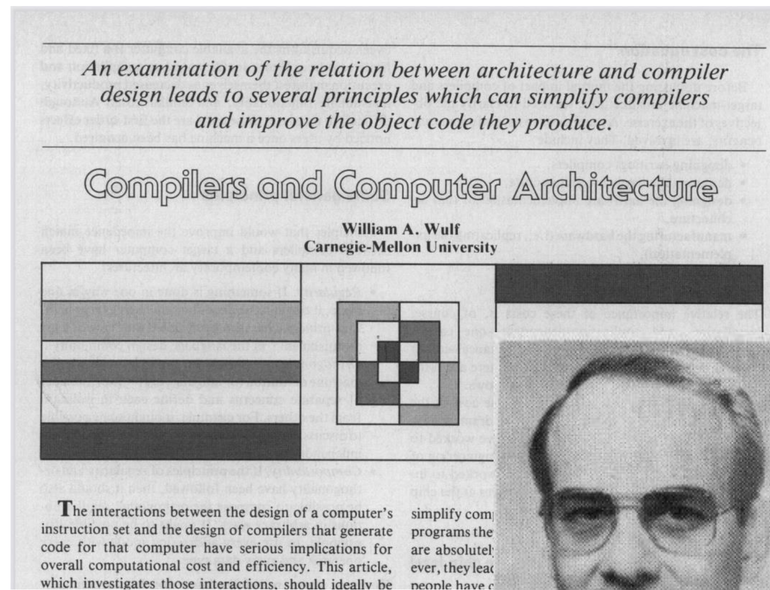
Regularity – “Law of least astonishment”
Orthogonality – keep separable concerns separate
Composability – regular, orthogonal ops combine easily

Specific Principles

One vs. All – precisely one way to do it, or all ways should be possible
Primitives, not solutions – solve by coding, compiling, & synthesizing

“Blatant opinions” (matters of taste)

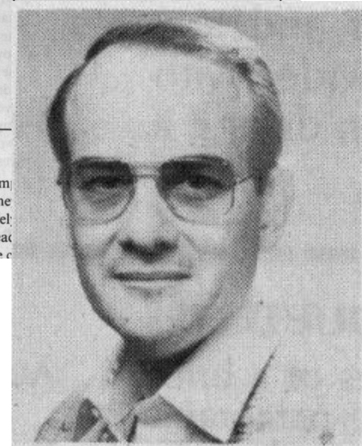
Addressing – not limited to simple arrays, etc.
Environment Support – exceptions, processes, debugging, etc
Deviations – deviate from these rules only in implementation-specific ways


An examination of the relation between architecture and compiler design leads to several principles which can simplify compilers and improve the object code they produce.

Compilers and Computer Architecture

William A. Wulf
Carnegie-Mellon University

The interactions between the design of a computer's instruction set and the design of compilers that generate code for that computer have serious implications for overall computational cost and efficiency. This article, which investigates those interactions, should ideally be



Designing irregular structures at the chip level is *very* expensive.

Some architectures have provided direct implementations of high-level concepts. In many cases these turn out to be more trouble than they are worth.

RISCV ISA

- We will learn about ISA design by learning about RISCV
- Modern, full-featured RISC ISA
- Developed in the last decade at UC Berkeley
  - The fifth in a sequence of RISC ISAs originating in the 80s
  - https://riscv.org/technical/specifications/
  - The RISC-V Instruction Set Manual, Volume I: BaseUser-Level ISA, Waterman et al, 2011
- Goals
  - Open-source
  - Free
  - Simple, but full-featured; avoids “over-architecting” for a particular uArch style (FPGA, ASIC,…)
  - Extensible through extension specifications and variants
  - Support heterogeneous & parallel systems efficiently
  - Support 32- and 64-bit variants efficiently
  - Fully virtualizable
  - Supports (but does not require) IEEE 754 Floating Point

Tooling

SPEC2017 Benchmark Suite

SPEC CPU® 2017

SPECspeed®2017 Integer	Language[1]	KLOC[2]	Application Area
600.perlbench_s	C	362	Perl interpreter
602.gcc_s	C	1,304	GNU C compiler
605.mcf_s	C	3	Route planning
620.omnetpp_s	C++	134	Discrete Event simulation - computer network
623.xalancbmk_s	C++	520	XML to HTML conversion via XSLT
625.x264_s	C	96	Video compression
631.deepsjeng_s	C++	10	Artificial Intelligence: alpha-beta tree search (Chess)
641.leela_s	C++	21	Artificial Intelligence: Monte Carlo tree search (Go)
648.exchange2_s	Fortran	1	Artificial Intelligence: recursive solution generator (Sudoku)
657.xz_s	C	33	General data compression

Pin and Pintools

Pin allows you to inspect and instrument an executable x86 binary, a single instruction at a time.

Why do we care?

Suppose when you execute the binary, you want it to stop every time there is a branch instruction, execute a different function (e.g. counting #branches), and then return back to the main binary.

Pin allows you to write a branch-counting pintool, which will “instrument” the original binary.

This means the pintool will add instructions to the original binary, causing it to call the branch-counting function every time there is a branch instruction. Pin then runs this modified, instrumented binary.

What is a pintool?

- **While Pin executes the instrumented binaries, the pintool defines how the binary is instrumented.**

```
pin -t $HOME/18344/obj-intel64/mem-count.so -o test.stats -- curl https://google.com
```

pintool (`-t $HOME/18344/obj-intel64/mem-count.so`)

binary to instrument (`curl https://google.com`)

The pintool specifies that an instrumentation function should be called for each instruction, which identifies the (load, store) instructions, and for each (load,store) instruction, calls another function to count the (load, store).

Important components of a pintool

We will focus on three main components of a pintool:

1. **Knobs**
   a. Command-line arguments to the pintool
2. **AddInstrumentFunction**
   a. Instrumenting each instruction
3. **AddFiniFunction**
   a. What runs after the main application is complete

Knobs - Command Line Arguments

Knobs allow passing command-line arguments to the pintool. This will be useful to dynamically select, e.g. a different branch prediction algorithm, from the command-line without recompiling the pintool.

Format of a Knob:

```
KNOB<datatype> KnobName ( KNOB_MODE, KNOB_FAMILY, PREFIX, DEFAULT_VALUE, PURPOSE );
```

datatype - type of the data being read from the command line (e.g. string, UINT32, etc)

KnobName - Name for the Knob, will be used to refer to the Knob through the rest of the program

KNOB_MODE - Indicates how multiple arguments for the same Knob are handled (e.g. if KNOB_MODE_WRITEONCE is used, only the first argument will be read into the Knob)

KNOB_FAMILY - Name for the family that the Knob belongs to, you can turn on/off Knobs by their families.

PREFIX - The flag that will be used on the command line for this Knob (e.g. -o for output file name)

DEFAULT_VALUE - Default value of the Knob if nothing specified on command line

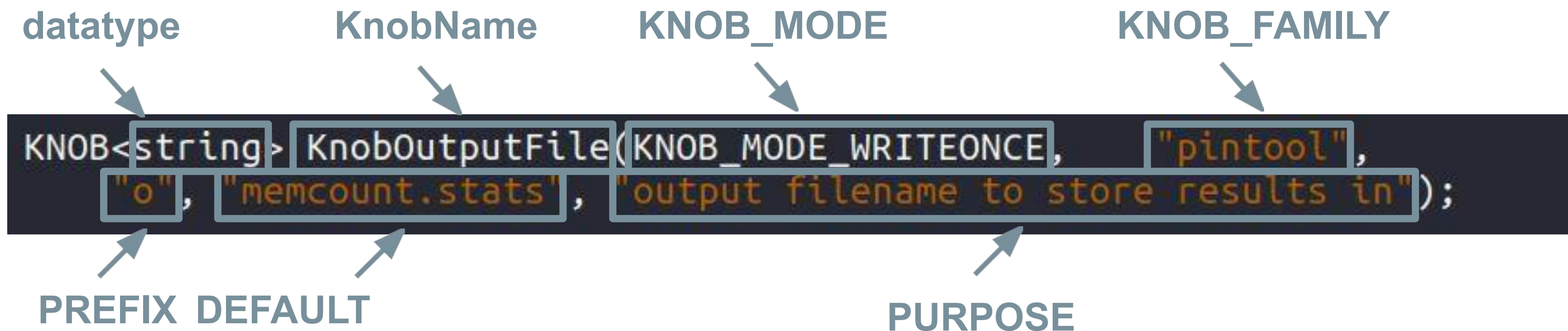
PURPOSE - String description that explains what the Knob does

Fields you will use in this class

Knobs - Command Line Arguments

```
KNOB<datatype> KnobName ( KNOB_MODE, KNOB_FAMILY, PREFIX, DEFAULT_VALUE, PURPOSE );
```

Example:



Passing a command-line argument to this Knob:

```
pin -t $HOME/18344/obj-intel64/mem-count.so -o test.stats -- curl https://google.com
```

Using Knobs in the pintool

```
KNOB<string> KnobOutputFile(KNOB_MODE_WRITEONCE,    "pintool",
    "o", "memcount.stats", "output filename to store results in");
```

The KnobName.Value() function gives the value stored in the Knob.

```
std::ofstream out(KnobOutputFile.Value().c_str());
```

Here, KnobOutputFile.Value() returns the string stored in the Knob. If nothing is specified on the command-line, this will return the default value, which is “memcount.stats” in this case.

You can define similar Knobs to read integer values, e.g. to specify the size of a cache, TLB, etc.

INS_AddInstrumentFunction

Allows you to specify a function that is called for every single instruction

```
// Insert call to function that runs for every instruction

INS_AddInstrumentFunction(Instruction, 0);
```

Here, `Instruction` is the function that is called for each instruction.

Example Instruction() - Instrument Mem. Accesses

Count Memory Operands in the instruction

If memory operand is read, call the Load() function.

If memory operand is written, call the Store() function.

We will provide you with the Instruction() required for the lab!

```
// Runs for every instruction

VOID Instruction(INS ins, void * v)
{
    UINT32 memOperands = INS_MemoryOperandCount(ins);

    // Instrument each memory operand. If the operand is both read and written
    // it will be processed twice.
    // Iterating over memory operands ensures that instructions on IA-32 with
    // two read operands (such as SCAS and CMPS) are correctly handled.
    for (UINT32 memOp = 0; memOp < memOperands; memOp++)
    {
        const UINT32 size = INS_MemoryOperandSize(ins, memOp);

        if (INS_MemoryOperandIsRead(ins, memOp))
        {
                // map sparse INS addresses to dense IDs
          const ADDRINT iaddr = INS_Address(ins);

          INS_InsertPredicatedCall(
                       ins, IPOINT_BEFORE,  (AFUNPTR) Load,
                       IARG_MEMORYOP_EA, memOp,
                       IARG_UINT32, size,
                       IARG_ADDRINT, iaddr,
                       IARG_END);
        }

        if (INS_MemoryOperandIsWritten(ins, memOp))
        {
          const ADDRINT iaddr = INS_Address(ins);

          INS_InsertPredicatedCall(
                       ins, IPOINT_BEFORE,  (AFUNPTR) Store,
                       IARG_MEMORYOP_EA,memOp,
                       IARG_UINT32, size,
                       IARG_ADDRINT, iaddr,
                       IARG_END);
        }
    }
}
```

Example Instruction() - Instrument Mem. Accesses

```
VOID Load(ADDRINT addr, UINT32 size, ADDRINT instAddr)
```

```
VOID Store(ADDRINT addr, UINT32 size, ADDRINT instAddr)
```

Here,

`addr` **- address being read/written to**

`size` **- size of the data being read/written**

`instAddr` **- address of the instruction itself**

For the labs, these arguments are all you need to know about. You will write functions that use these arguments for solving the labs.

We will provide you with the Instruction() required for the lab!

```
// Runs for every instruction

VOID Instruction(INS ins, void * v)
{
    UINT32 memOperands = INS_MemoryOperandCount(ins);

    // Instrument each memory operand. If the operand is both read and written
    // it will be processed twice.
    // Iterating over memory operands ensures that instructions on IA-32 with
    // two read operands (such as SCAS and CMPS) are correctly handled.
    for (UINT32 memOp = 0; memOp < memOperands; memOp++)
    {
        const UINT32 size = INS_MemoryOperandSize(ins, memOp);

        if (INS_MemoryOperandIsRead(ins, memOp))
        {
                // map sparse INS addresses to dense IDs
          const ADDRINT iaddr = INS_Address(ins);

          INS_InsertPredicatedCall(
                       ins, IPOINT_BEFORE,  (AFUNPTR) Load,
                       IARG_MEMORYOP_EA, memOp,
                       IARG_UINT32, size,
                       IARG_ADDRINT, iaddr,
                       IARG_END);
        }

        if (INS_MemoryOperandIsWritten(ins, memOp))
        {
          const ADDRINT iaddr = INS_Address(ins);

          INS_InsertPredicatedCall(
                       ins, IPOINT_BEFORE,  (AFUNPTR) Store,
                       IARG_MEMORYOP_EA,memOp,
                       IARG_UINT32, size,
                       IARG_ADDRINT, iaddr,
                       IARG_END);
        }
    }
}
```

Example Instruction() - Instrument Branch Instructions

Check if instruction is a branch

TRUE for only conditional branches

```
void InstrumentInstruction(INS ins, void *v) {
  if (INS_IsBranch(ins) && INS_HasFallThrough(ins)) {
    INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR) branch,
        IARG_INST_PTR, IARG_BRANCH_TAKEN, IARG_END);
  }
}
```

```
void branch(ADDRINT pc, bool brTaken)
```

Here,

`pc` **- address of the instruction**

`brTaken` **- whether the branch is taken or not**

We will provide you with the Instruction() required for the lab!

INS_AddFiniFunction

Allows you to specify a function that is called after the original binary has completed execution

```
INS_AddFiniFunction(Fini, 0);
```

```
VOID Fini(INT32 code, VOID * v)
{
    std::ofstream out(KnobOutputFile.Value().c_str());

    //Output your results here

    out.close();
}
```

tmux demo

Q&A + impromptu OH