CS2002 Logic and Architecture
Lecture 2
ISA design and Compilation

Ian Gent

Last time

- Books for Architecture
- Aims
- History
- Stored programs
- von Neumann architecture
- Fetch-execute cycle
- ISA
- Example x86 instruction
- Machine Code
- Assembly Language

This time

- Issues in ISA design:
  - registers
  - operands
  - memory addressing
  - CISC vs RISC
- Compilation process:
  - compiler
  - assembler
  - linker
  - loader

Some key issues in ISA design

- Registers
- Operands
- Memory addressing
- CISC versus RISC designs

Registers

A key part of understanding both:

1. ISAs
2. assembly language

Ways to think about registers

Some kind of magic location on CPU

modern CPUs no longer have number of registers you think they have

The very fastest form of memory

much faster than L1 cache

The only form of local variable you have in assembly language

only a small number allowed

you can't choose their names or control their scope

Kinds of registers

General-purpose registers store one word (e.g. 64 bit) and are used in integer operations

typically a small number, e.g. x86 has 16 such registers

often possible to use part of a register to access smaller units such as 16 bit words or octets

Floating-point registers cf. IEEE Standard 754

Special registers such as:

- **instruction pointer** (*ip*) a.k.a. **program counter** (*pc*)
- flags

Kinds of registers (cont.)

In some architectures, some registers are reserved for specific purposes, e.g., to hold memory addresses

Most ISAs that are considered to be “orthogonal” don’t distinguish between registers

i.e. if you can do something with one register you can do it with any register

ARM architecture is orthogonal, x86 historically isn’t

Register allocation

Moving data to/from memory is expensive (more than doing the actual operations on data)

better to avoid writing intermediate results back to memory

But number of registers is limited

care needed that registers are not overwritten by subroutines

Conventions help, will return to this later

Operands and results

Each ISA instruction performs some operation

e.g. load, store, add, multiply, branch

ISA specifies operations and their **operands**

(**operands** = things the CPU can apply operations to, like registers, memory)

Where can arguments come from and results be stored to?

These are critical ISA design decisions

Operands and results (cont.)

Register-memory

- at least one argument can come straight from memory
- more expressive, less need to find registers to store things in

Load-store (a.k.a. **register-register**)

- all operations work between registers
- only data transfer operations access memory
- separation of memory access and arithmetic functions makes design simpler

How many operands does each operation take?

Three:

- operation specifies both inputs and an output location
- e.g. “add x to y, storing result in z”
- instructions get longer to specify operands, more memory access to load them

Two:

- result replaces one input
- e.g. “add x to y, changing y”
- or goes to dedicated register (rare)

How many operands does each operation take? (cont.)

One (rare):

- all operations modify an “accumulator”
- e.g. “add x to the accumulator”

Zero (even rarer):

- e.g. “replace the top two numbers of the stack by their sum”

Examples

Different ways to do C = A + B with different operand models with C, A, B in memory

stack	accum	reg-mem	load-store 2 arg	load-store 3 arg
push A	load A	load r1, A	load r1, A	load r1, A
push B	add B	add r1, B	load r2, B	load r2, B
add	store C	store C, r1	add r1, r2	add r3, r1, r2
pop C			store C, r2	store C, r3

Rough trade-off (but at least two dimensions are involved here):

short instructions ⟷ flexibility

How many bits should a single address refer to?

- if few, then address is long
- if many, then extra data is moved, e.g. by ‘load’ instruction

Word addressed – one address for each word

- extra info to refer to specific byte within word

Byte addressed – one address for each 8 bit byte (octet)

- lowest address in a word stands for the word
- now standard
- e.g. 32-bit (4-byte) word would typically start at (byte) address that is multiple of 4
- if not (**misaligned**), then disallowed or slower

Big-endian and little-endian

Names due to Jonathan Swift, **Gulliver's Travels** (1726)

Suppose we have number **0xDEADBEEF** (hexadecimal)

(remember **A**=10, . . . , **F**=15)

stored in 4 bytes at addresses 0, 1, 2, 3

Then depending on **endianness**:

DE	**AD**	**BE**	**EF**
0	1	2	3

big-endian

EF	**BE**	**AD**	**DE**
0	1	2	3

little-endian

Memory addressing modes

How does ISA allow instructions to access memory?

immediate mode

- value is specified in instruction itself
- used for constants

direct mode

- value is at fixed address specified in instruction
- used for ‘global variables’

Memory addressing modes (cont.)

indirect mode

- value is at fixed address that is in register
- extra power relative to immediate and direct modes
- but something like `a = b[i]` is cumbersome to implement

index mode

- value is at fixed address plus index obtained from register
- allows implementation of `a = b[i]` in one instruction

This is not an exhaustive list of addressing modes . . .

Any ISA allows some set of addressing modes

CISC and RISC

Trade-offs in ISA design:

- orthogonality simplifies code development
- with more powerful instructions, fewer instructions are required to do the same thing

But more orthogonality and more powerful instructions entail:

- chips may need to be bigger and more complex
- slower if multiple memory accesses done in single instruction
- or if calculations have to wait for memory access
- longer instructions if more operands

CISC and RISC (cont.)

Roughly up to 1985:

- space on chips was very limited
- lot of assembly language programming done by humans
- memory to store programs was limited

So **CISC** (**complex instruction set computer**):

- barely possible to implement multiplication of two dedicated registers, but not for all (no orthogonality)
- lots of addressing modes designed for human programmers, and adding **BCD** (**binary coded decimal**)
- variable-length instructions (complex ones long, simple ones short)

CISC and RISC (cont.)

Later:

- transistors got smaller, more room on chip
- compilers got better, machine-generated assembly
- memory was bigger, more space for storing programs
- CPUs started *pipelining* more than one instruction at once
  - while executing one instruction, already fetch next
  - multiple stages of execution
  - much easier if all instructions are same length and each stage takes same time

So **RISC** (**reduced instruction set computer**):

- most instructions have three operands
- many registers
- often register-register, and few addressing modes

CISC and RISC (cont.)

With these RISC designs:

- more instructions done per second
- but more instructions needed for any task

Early RISC designs suffered from large size of machine code requiring lots of instructions to be loaded

CISC vs RISC: who won?

There was definitely a winner: **RISC**

But the name is bad: a modern RISC ISA may have many more instructions than an old CISC ISA

The key difference is that modern RISC is a load-store architecture

Modern RISC designs trade bit of elegance for compactness, and bit of simplicity for compatibility

Modern CISC designs translate instructions *on the chip* into several RISC-like “micro-ops”

This includes x86-64, which is still a CISC ISA but implemented as RISC

Compilers and interpreters

Given CPU that understands certain ISA

how to run program written in Java, C, Haskell or Python?

Compiler converts high-level program into low level instructions for specific machine

not (usually) needed at run time

Interpreter is a program that reads a high-level program and carries it out

done at run time

Assemblers, linkers

Compilers often actually create assembly code

e.g. clang, gcc do this

Assembler converts **assembly language** for ISA to more detailed **object code**

think of it as compiler for low level language

where that language is based on ISA of intended hardware

Linker combines object code files together to make an executable program that can actually run

incorporating necessary library object code files

and determining memory offsets for functions in different files

Operating system

After:

- compiling
- assembling
- linking

the next step is:

- loading

The **loader** loads programs and libraries into memory to prepare them for execution

Provided by the **operating system** (OS)

which also provides access to hardware resources

Examples with clang

Normal way to compile simple program to create executable foo:

```
clang foo.c -o foo
```

This is the default behaviour so you don't realise how many stages are involved

But this is multiple stages in one and we can break it up

Examples with clang (cont.)

Do C preprocessor only, create new C file with e.g. header files included:

```
clang -E foo.c -o foo-preprocessed.c
```

Compile preprocessed file, create assembler file `foo.s`:

```
clang -S foo-preprocessed.c -o foo.s
```

Assemble the assembly code. Create object code file `foo.o`:

```
clang -c foo.s -o foo.o
```

Link object code `foo.o` and create executable `foo`:

```
clang foo.o -o foo
```

Exercise

Try out the different stages of compilation with clang, e.g. using the following C program in `foo.c`:

```
#include <stdio.h>
int main() {
    int rows, i;
    printf("Enter number of rows: ");
    scanf("%d",&rows);
    for(i=0; i<rows; ++i)
        printf("Hello, World!\n");
    return 0;
}
```

Recommended reading

Comer:

- Section 4.12 (preprocessing)
- Sections 5.1–5.6, 7.1–7.6 (ISAs)