CS2002 Logic and Architecture
Lecture 3
CISC and RISC

Ian Gent

Last time

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

Recommended reading from last time

Comer:

- Section 3.12 (endianness)
- Sections 3.19–3.20 (BCD)
- Sections 5.7, 5.9 (general-purpose registers)
- Section 5.11 (CISC/RISC)
- Sections 7.7–7.14 (addressing modes)

This time

- Introduction to the x86-64 ISA:
  - registers
  - flags
  - memory addressing
- Looking at a compiled assembly file

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;
}
```

Good reasons to teach assembly language

- It gives you an insight into computer architecture because there is a match between assembly instructions and hardware
- It shows you how statements in a high level language are broken down for execution on an actual CPU
- It helps you understand how compilers work because you can understand the output of (e.g.) clang
- It *especially* helps you understand how function calling works
- You may need to use a few lines of assembly to get at special-purpose optimised instructions for the specific CPU

Why Teach x86?

Not in the syllabus but labs are full of x86 machines

But we used to teach *MIPS*, a lovely orthogonal RISC architecture

And easy to emulate

So why did we change to x86?

It's a good idea to teach on active ISAs

They are under active development

Toolchains for using it are up to date

It is (was?) the most widely used ISA on modern computers/servers

ARM dominates phones/tablets

x86-64 ISA

Intel 64 and IA-32 Architectures Software Developer's Manuals

`https://software.intel.com/en-us/articles/intel-sdm`

In several volumes, altogether 5066 pages (November 2020)

Not compulsory reading for this module!

We can only cover the main points, for now:

- registers

Our focus will be on the 64 bit version

General Purpose Registers in x86-64

16 registers of 64 bits

8 historical names
8 numerical names

But wait, there's more!

64 bit registers
`%rax`
`%rbx`
`%rcx`
`%rdx`
`%rsi`
`%rdi`
`%rbp`
`%rsp`
`%r8`
`%r9`
`%r10`
`%r11`
`%r12`
`%r13`
`%r14`
`%r15`

General Purpose Registers in x86-64 (cont.)

Low 32 bits addressable
So 16 GPR of 32 bits

First 8 are %e**
e for Extended
Next 8 are %r**d
d for Double

Changing %eax *also* changes %rax

But wait, there's more!

64 bit registers	low 32 bits
%rax	%eax
%rbx	%ebx
%rcx	%ecx
%rdx	%edx
%rsi	%esi
%rdi	%edi
%rbp	%ebp
%rsp	%esp
%r8	%r8d
%r9	%r9d
%r10	%r10d
%r11	%r11d
%r12	%r12d
%r13	%r13d
%r14	%r14d
%r15	%r15d

General Purpose Registers in x86-64 (cont.)

Low 16 bits addressable
So 16 GPR of 16 bits each

First 8, just delete e/r
Second 8, replace d with w for Word

Changing %ax *also* changes %eax %rax

But wait, there's more!

64 bit registers	low 32 bits	low 16 bits
`%rax`	`%eax`	`%ax`
`%rbx`	`%ebx`	`%bx`
`%rcx`	`%ecx`	`%cx`
`%rdx`	`%edx`	`%dx`
`%rsi`	`%esi`	`%si`
`%rdi`	`%edi`	`%di`
`%rbp`	`%ebp`	`%bp`
`%rsp`	`%esp`	`%sp`
`%r8`	`%r8d`	`%r8w`
`%r9`	`%r9d`	`%r9w`
`%r10`	`%r10d`	`%r10w`
`%r11`	`%r11d`	`%r11w`
`%r12`	`%r12d`	`%r12w`
`%r13`	`%r13d`	`%r13w`
`%r14`	`%r14d`	`%r14w`
`%r15`	`%r15d`	`%r15w`

General Purpose Registers in x86-64 (cont.)

Low 8 bits addressable
So 16 GPR of 8 bits each

First 4, replace x by l for Low
Second 4, add l for Low
Third 8, replace w with b for Byte

Changing %al *also* changes %ax %eax %rax

But wait, there's more!

64 bit registers	low 32 bits	low 16	low 8 bits
%rax	%eax	%ax	%al
%rbx	%ebx	%bx	%bl
%rcx	%ecx	%cx	%cl
%rdx	%edx	%dx	%dl
%rsi	%esi	%si	%sil
%rdi	%edi	%di	%dil
%rbp	%ebp	%bp	%bpl
%rsp	%esp	%sp	%spl
%r8	%r8d	%r8w	%r8b
%r9	%r9d	%r9w	%r9b
%r10	%r10d	%r10w	%r10b
%r11	%r11d	%r11w	%r11b
%r12	%r12d	%r12w	%r12b
%r13	%r13d	%r13w	%r13b
%r14	%r14d	%r14w	%r14b
%r15	%r15d	%r15w	%r15b

General Purpose Registers in x86-64 (cont.)

Four more 8 bit GPR

The high byte of the low 16 bits of the first four registers

Replace l with h for High

Changing %ah *also* changes %ax %eax %rax

Where on earth are we?

64 bit registers	low 32 bits	low 16	low 8 bits
`%rax`	`%eax`	`%ax/ah`	`%al`
`%rbx`	`%ebx`	`%bx/bh`	`%bl`
`%rcx`	`%ecx`	`%cx/ch`	`%cl`
`%rdx`	`%edx`	`%dx/dh`	`%dl`
`%rsi`	`%esi`	`%si`	`%sil`
`%rdi`	`%edi`	`%di`	`%dil`
`%rbp`	`%ebp`	`%bp`	`%bpl`
`%rsp`	`%esp`	`%sp`	`%spl`
`%r8`	`%r8d`	`%r8w`	`%r8b`
`%r9`	`%r9d`	`%r9w`	`%r9b`
`%r10`	`%r10d`	`%r10w`	`%r10b`
`%r11`	`%r11d`	`%r11w`	`%r11b`
`%r12`	`%r12d`	`%r12w`	`%r12b`
`%r13`	`%r13d`	`%r13w`	`%r13b`
`%r14`	`%r14d`	`%r14w`	`%r14b`
`%r15`	`%r15d`	`%r15w`	`%r15b`

General Purpose Registers in x86-64 (cont.)

Total $16 \times 4 + 4 = 68$ GPR

But they overlap in many ways

Names are very inconsistent

To keep backwards compatibility

There's special purpose registers too ...

64 bit registers	low 32 bits	low 16	low 8 bits
%rax	%eax	%ax/ah	%al
%rbx	%ebx	%bx/bh	%bl
%rcx	%ecx	%cx/ch	%cl
%rdx	%edx	%dx/dh	%dl
%rsi	%esi	%si	%sil
%rdi	%edi	%di	%dil
%rbp	%ebp	%bp	%bpl
%rsp	%esp	%sp	%spl
%r8	%r8d	%r8w	%r8b
%r9	%r9d	%r9w	%r9b
%r10	%r10d	%r10w	%r10b
%r11	%r11d	%r11w	%r11b
%r12	%r12d	%r12w	%r12b
%r13	%r13d	%r13w	%r13b
%r14	%r14d	%r14w	%r14b
%r15	%r15d	%r15w	%r15b

General-purpose registers in x86 with 64 bits

There are (only!) 16 general-purpose 64-bit registers

So $16 \cdot 8 = 128$ bytes of the fastest storage in a modern ISA

x86-32 had 8 32-bit registers

When 64-bit architecture was introduced:

- registers doubled in length from 32 to 64 bits
- and number doubled from 8 to 16

New registers are called:

r8 r9 r10 r11 r12 r13 r14 r15

The names of the extended versions of old registers were:

rax rbx rcx rdx rsi rdi rbp rsp

General-purpose registers in x86 with 64 bits (cont.)

Where do these names come from?

Originally from Intel 8086 (1978), which had:

- 4 16-bit registers *AX BX CX DX*
  - *AX* was (primary) **accumulator**, some instructions could only use this register
  - *BX* was ‘base’ accumulator
  - *CX* was **counter**, some instructions intended for loops could only use this
  - *AL* and *AH* were low and high bytes of *AX*, etc.
- two index registers for memory access: *SI* (**source index**) and *DI* (**destination index**)
- *SP* was the **stack pointer**, *BP* something similar called a **base pointer**

General-purpose registers in x86 with 64 bits (cont.)

Then Intel 80386 (1985) introduced 32-bit version

More orthogonal

Registers now called:

EAX EBX ECX EDX ESI EDI EBP ESP

with *E* for ‘extended’

Then came the 64-bit version (2000)

with *r* now simply for ‘register’ (with say *rax* instead of *r1* for historical reasons)

Similar story for two special-purpose registers:

- *rip* (instruction pointer, a.k.a. program counter)
- *rflags* (binary flags, e.g. for comparisons)

Flags register

Let us look more closely at the *rflags* register:

- 64 bits, but not available as general purpose register
- individual bits reflect execution of preceding instruction
- conditional jumps executed based on these flags

E.g.

- bit **0** is **carry flag** (did add/sub result in carry bit?)
- bit **2** is **parity flag** (last result had even number of bits = 1 ?)
- bit **6** is **zero flag** (last result was zero?)
- bit **7** is **sign flag** (last result was negative?)
- bit **11** is **overflow flag** (did add/sub result in overflow?)

More information:

`https://en.wikipedia.org/wiki/FLAGS_register`

Reminder - Memory Addressing

There are a number of ways you can give operands values

Two don’t involve memory at all

- **Registers**
  Just give an argument as a register
- **Immediate mode**
  value is specified in instruction itself
  used for constants that can be fixed at compile/linking time

Be careful:

“Immediate” is often called a “memory addressing mode” but actually it doesn’t address memory

The value encoded in the instruction is used - hence “immediate”

Reminder - Memory Addressing (cont.)

Other addressing modes get values stored in memory

- **Direct mode**
  Memory address encoded in the instruction
- **Indirect mode**
  Memory address is stored in a register
- **Index mode**
  Memory address is a given value plus an offset determined by a register

Memory Addressing in x86-64

x86-64 only has **three** addressing modes

(Two if you don’t count immediate)

- **Immediate**
- **General**
- **RIP Relative Addressing**

We don’t need to talk much about immediate, so will look at the others.

Memory addressing in x86

Main addressing mode in x86 is fairly general (AT&T syntax):

disp`(%`*reg1*`,%`*reg2*`,`*scale*`)`

- *disp* is constant integer (**displacement**, a.k.a. **offset**)
- *reg1* is the **base** register of the operand
- *reg2* is the **index** register of the operand
- *scale* can only be 1,2,4,8

Refers to address:

(*reg1*) + *scale* * (*reg2*) + *disp*

where (*reg1*) and (*reg2*) denote the values in the two registers

Special Cases of General Mode

We can use this one mode to get many others by careful omission

General:

disp `(%`*reg1*`, %`*reg2*`,` *scale*`)`

E.g. we can omit displacement (default 0) and scale (default 1):

`(%`*reg1*`, %`*reg2*`)`

E.g. we can omit displacement and *reg1*:

`(, %`*reg2*`,` *scale*`)`

Special Cases of General Mode (cont.)

Why don’t we have modes like direct/indirect/index?

We do!

Just omit the rights bits of the general case

Direct mode:

disp

Indirect mode:

`(%`*reg1*`)`

Index mode: (based on a register)

`(%`*reg1*`,%`*reg2*`,`*scale*`)`

Index mode: (based on a direct address)

disp`(,%`*reg2*`,`*scale*`)`

RIP relative addressing

RIP relative addressing was introduced with x86-64

disp (% *rip*)

Avoids absolute address required by direct addressing mode

Instead address is current value of *rip* plus displacement

(Current value of *rip* is of instruction **following** current instruction)

Advantage of RIP rel. addressing: **position-independent code**

Code that runs no matter where it is loaded in memory

Disadvantage: displacement is limited to 32 bits

From C to assembly

Example file `sam.c` :

```
long sample(long x) {
    static long total;
    total += x;
    return total;
}
```

Compiled to assembly file `sam.s` with:

```
clang -S sam.c -o sam.s \
     -fno-verbose-asm -fomit-frame-pointer
```

- `-S` produces assembly code instead of object code
- don't worry about other flags

The whole assembly file

```
        .text
        .file   "sam.c"
        .globl  sample
        .align  16, 0x90
        .type   sample,@function
sample:
        .cfi_startproc
        movq    %rdi, -8(%rsp)
        movq    -8(%rsp), %rdi
        addq    sample.total, %rdi
        movq    %rdi, sample.total
        movq    sample.total, %rax
        retq
.Lfunc_end0:
        .size   sample, .Lfunc_end0-sample
        .cfi_endproc
        .type   sample.total,@object
        .local  sample.total
        .comm   sample.total,8,8
        .ident  "clang version 3.8.0-2ubuntu4 (tags/ etc. "
        .section        ".note.GNU-stack","",@progbits
```

The actual x86 instructions

```
     .text
     .file   "sam.c"
     .globl  sample
     .align  16, 0x90
     .type   sample,@function
sample:
     .cfi_startproc
     movq    %rdi, -8(%rsp)
     movq    -8(%rsp), %rdi
     addq    sample.total, %rdi
     movq    %rdi, sample.total
     movq    sample.total, %rax
     retq
.Lfunc_end0:
     ...
```

What is the rest?

Assembler labels

```
        .text
        .file   "sam.c"
        .globl  sample
        .align  16, 0x90
        .type   sample,@function
sample:
        .cfi_startproc
        movq    %rdi, -8(%rsp)
        movq    -8(%rsp), %rdi
        addq    sample.total, %rdi
        movq    %rdi, sample.total
        movq    sample.total, %rax
        retq
.Lfunc_end0:
        ...
```

Names for places in program

E.g. for jumps

Turned into addresses by assembler and/or linker

Assembler directives

```
    .text
    .file   "sam.c"
    .globl  sample
    .align  16, 0x90
    .type   sample,@function
sample:
    .cfi_startproc
    movq     %rdi, -8(%rsp)
    ...
    retq
.Lfunc_end0:
    .size   sample, .Lfunc_end0-sample
    .cfi_endproc

    .type   sample.total,@object
    .local  sample.total
    ...
```

Meant for assembler/linker

Assembler directives (cont.)

```
.text
.file  "sam.c"
.globl  sample
.align  16, 0x90
. . .
```

- **.text** defines current section as text (not data)
- **.file** where code comes from
- **.globl** identifiers in partial program
  - to be made available to other partial programs
  - here only `sample`
- **.align** how next data is to be aligned
  - here next data should be at multiple of 16
  - padding with bytes 0x90

Assembler directives (cont.)

```
    ...
    .type   sample,@function
sample:
    .cfi_startproc
    ...
    .size sample, .Lfunc_end0-sample
    .cfi_endproc
    .type   sample.total,@object
    ...
```

- **.type** declares type of symbol
- **.cfi_ ...** used for debugging
- **.size** size of symbol, used by debugger

Assembler directives (cont.)

```
...
.local  sample.total
.comm   sample.total,8,8
.ident  "clang version 3.8.0-2 etc. "
.section ".note.GNU-stack","",@progbits
```

- **.local** says symbol not externally visible
- **.comm** declares common symbol of length, with alignment
- **.ident** used to place tags in object files
- **.section** new section is created

Next time

- x86:
  - dissecting an assembly file
  - Calling Conventions
  - Stack Frames
  - function calling