CS429: Computer Organization and Architecture

Instruction Set Architecture

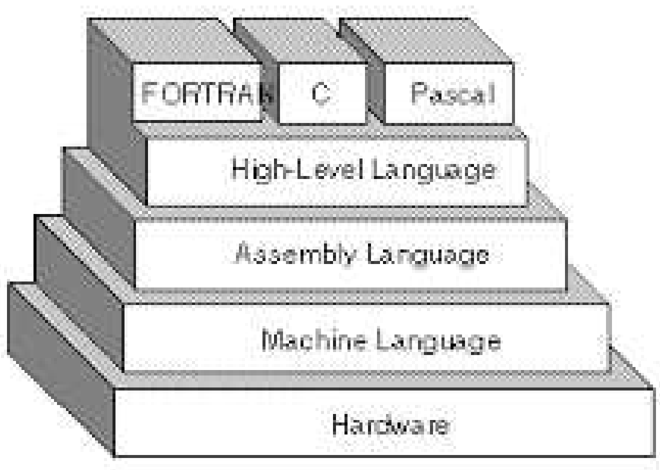
Dr. Bill Young
Department of Computer Science
University of Texas at Austin

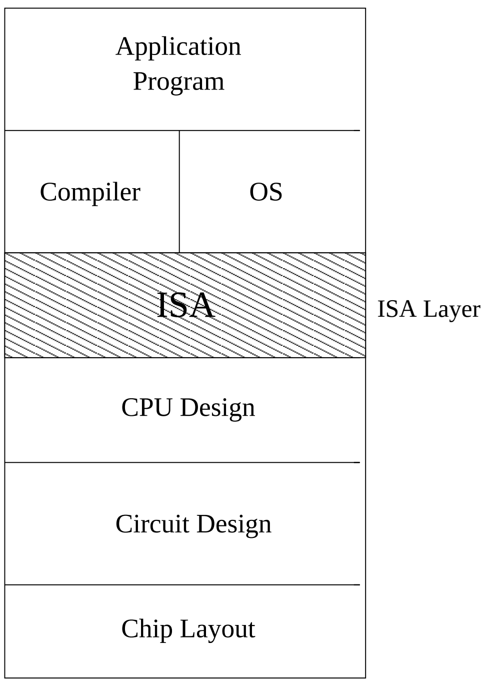
Last updated: October 2, 2019 at 18:05

Topics of this Slideset

- Intro to Assembly language
- Programmer visible state
- Y86 Rudiments
- RISC vs. CISC architectures

Instruction Set Architecture





Assembly Language View

- Processor state: registers, memory, etc.
- Instructions and how instructions are encoded

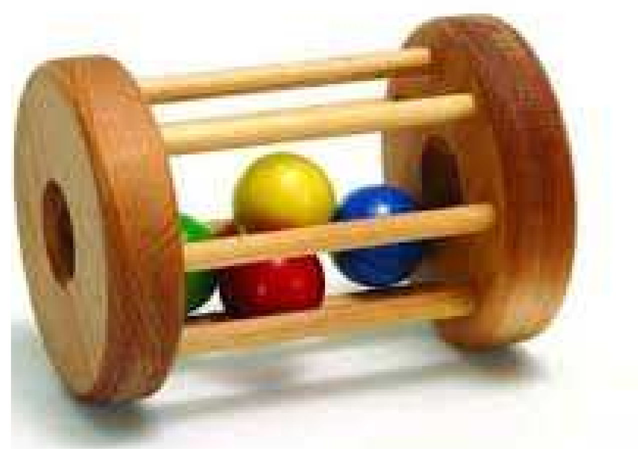
Layer of Abstraction

- Above: how to program machine, processor executes instructions sequentially
- Below: What needs to be built
  - Use variety of tricks to make it run faster
  - E.g., execute multiple instructions simultaneously

Why Y86?

The Y86 is a "toy" machine that is similar to the x86 but *much simpler*. It is a gentler introduction to assembly level programming than the x86.

- just a few instructions as opposed to hundreds for the x86;
- fewer addressing modes;
- simpler system state;
- absolute addressing.



Everything you learn about the Y86 will apply to the x86 with very little modification. But the main reason we're bothering with the Y86 is because we'll be explaining pipelining in that context.

Language / Machine Semantics

There are various means of giving a *semantics* or meaning to a programming system.

Probably the most sensible for an assembly (or machine) language is an *operational semantics*, also known as an *interpreter semantics*.

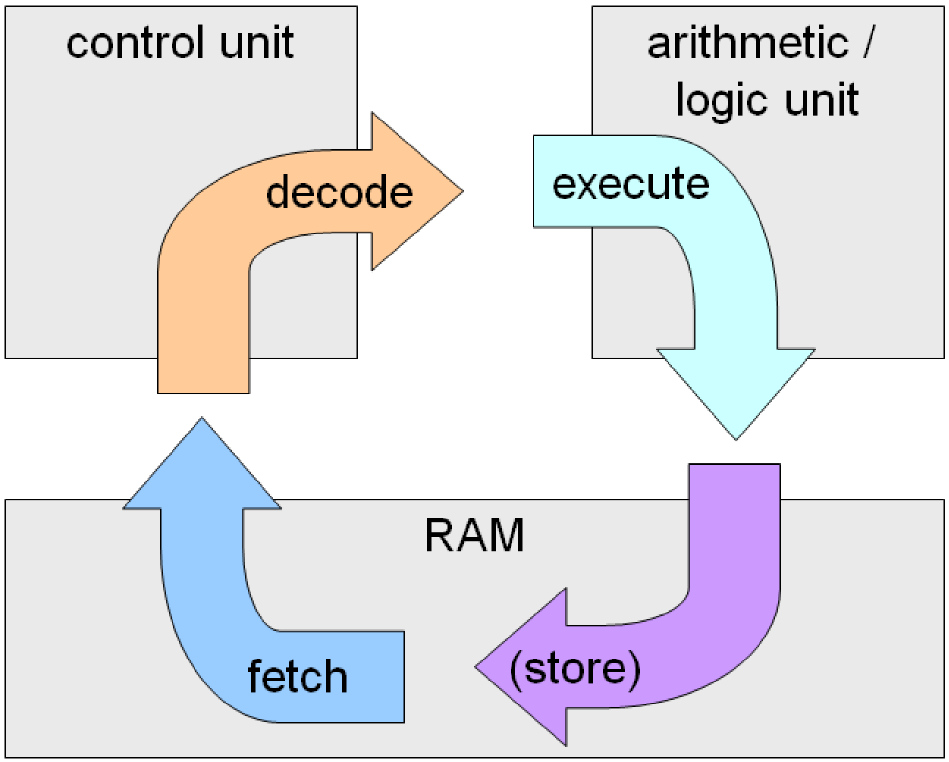
That is, we explain the semantics of each possible operation in the language by explaining the effect that execution of the operation has on the *machine state*.

Fetch / Decode / Execute Cycle

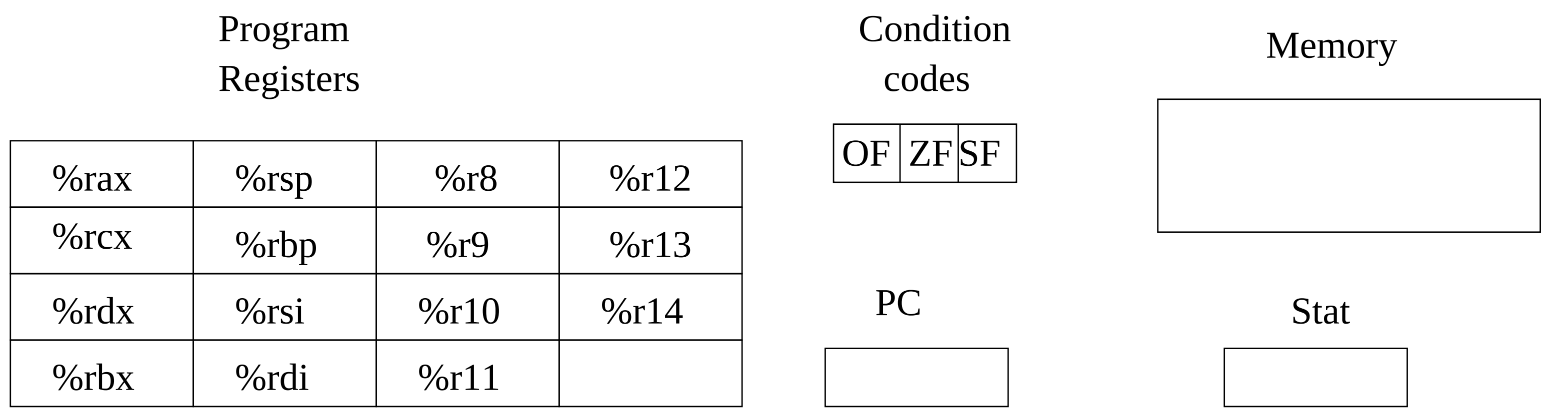
The most fundamental abstraction for the machine semantics for the x86/Y86 or similar machines is the *fetch-decode-execute* cycle. This is also called the *von Neumann architecture*.

The machine repeats the following steps forever:

1. fetch the next instruction from memory (the PC tells you which is next);
2. decode the instruction (in the control unit);
3. execute the instruction, updating the state appropriately;
4. go to step 1.



Y86 Processor State



- Program registers: almost the same as x86-64, each 64-bits
- Condition flags: 1-bit flags set by arithmetic and logical operations. OF: Overflow, ZF: Zero, SF: Negative
- Program counter: indicates address of instruction
- Memory
  - Byte-addressable storage array
  - Words stored in little-endian byte order
- Status code: (status can be AOK, HLT, INS, ADR) to indicate state of program execution.

Y86 Instructions

We're actually describing two languages: the assembly language and the machine language. There is nearly a 1-1 correspondence between them.

Machine Language Instructions

- 1-10 bytes of information read from memory
  - Can determine instruction length from first byte
  - Not as many instruction types and simpler encoding than x86-64
- Each instruction accesses and modifies some part(s) of the program state.

Y86 Instruction Set

Byte	0	1	2	3	4	5	6	7	8	9
`halt`	0 0									
`nop`	1 0									
`cmovXX rA,rB`	2 fn	rA rB								
`irmovq V,rB`	3 0	F rB	V							
`rmmovq rA,D(rB)`	4 0	rA rB	D							
`mrmovq D(rB),rA`	5 0	rA rB	D							
`OPq rA,rB`	6 fn	rA rB								
`jXX Dest`	7 fn	Dest								
`call Dest`	8 0	Dest								
`ret`	9 0									
`pushq rA`	A 0	rA F								
`popq rA`	B 0	rA F								

Example from C to Assembly

Suppose we have the following simple C program in file `code.c`.

```
int sumInts(long int n)
{
  /* Add the integers from 1..n. */
  long int i;
  long int sum = 0;
  for ( i = 1; i <= n; i++ ) {
    sum += i;
  }
  return sum;
}
```

We used `long int` to force usage of the 64-bit registers. You can generate assembly using the following command:

```
> gcc -O -S code.c
```

x86 Assembly Example

```
        .file   "code.c"
        .text
        .globl  sumInts
        .type   sumInts, @function
sumInts:
.LFB0:
        .cfi_startproc
        testq   %rdi, %rdi
        jle     .L4
        movq    $0, %rax
        movq    $1, %rdx
.L3:
        addq    %rdx, %rax
        addq    $1, %rdx
        cmpq    %rdx, %rdi
        jge     .L3
        ret
.L4:
        movq    $0, %rax
        ret
        .cfi_endproc
.LFE0:
        .size   sumInts, .-sumInts
        .ident  "GCC: (Ubuntu 4.8.4-2ubuntu1~14.04) 4.8.4"
```

Y86 Assembly Example

This is a hand translation into Y86 assembler:

```
sumInts:
        andq   %rdi, %rdi              # test %rdi = n
        jle    .L4                     # if <= 0, done
        irmovq $1, %rcx                # constant 1
        irmovq $0, %rax                # sum = 0
        irmovq $1, %rdx                # i = 1
.L3:
        rrmovq %rdi, %rsi              # temp = n
        addq   %rdx, %rax              # sum + = i
        addq   %rcx, %rdx              # i += 1
        subq   %rdx, %rsi              # temp -= i
        jge    .L3                     # if >= 0, goto L3
        ret                            # else return sum
.L4:
        irmovq $0, %rax                # done
        ret
```

How does it get the argument? How does it return the value?

Encoding Registers

Each register has an associated 4-bit ID:

%rax	0	%r8	8
%rcx	1	%r9	9
%rdx	2	%r10	A
%rbx	3	%r11	B
%rsp	4	%r12	C
%rbp	5	%r13	D
%rsi	6	%r14	E
%rdi	7	no reg	F

Almost the same encoding as in x86-64.

Most of these registers are general purpose; `%rsp` has special functionality.

Y86 Instruction Set (2)

`cmovXX rA,rB`	2	fn	rA	rB

Encompasses:

`rrmovq rA,rB`	2	0	move from register to register
`cmovle rA,rB`	2	1	move if less or equal
`cmovl rA,rB`	2	2	move if less
`cmove rA,rB`	2	3	move if equal
`cmovne rA,rB`	2	4	move if not equal
`cmovge rA,rB`	2	5	move if greater or equal
`cmovg rA,rB`	2	6	move if greater

Y86 Instruction Set (3)

`OPq rA,rB`	6	fn	rA	rB

Encompasses:

`addq rA,rB`	6	0	add
`subq rA,rB`	6	1	subtract
`andq rA,rB`	6	2	and
`xorq rA,rB`	6	3	exclusive or

Y86 Instruction Set (4)

`jXX Dest`	7	fn	Dest

Encompasses:

`jmp Dest`	7	0	unconditional jump
`jle Dest`	7	1	jump if less or equal
`jl Dest`	7	2	jump if less
`je Dest`	7	3	jump if equal
`jne Dest`	7	4	jump if not equal
`jge Dest`	7	5	jump if greater or equal
`jg Dest`	7	6	jump if greater

Simple Addressing Modes

- **Immediate:** `value`

  `irmovq  $0xab, %rbx`

- **Register:** `Reg[R]`

  `rrmovq  %rcx, %rbx`

- **Normal (R):** `Mem[Reg[R]]`
  - Register R specifies memory address.
  - This is often called *indirect* addressing.

  `mrmovq (%rcx), %rax`

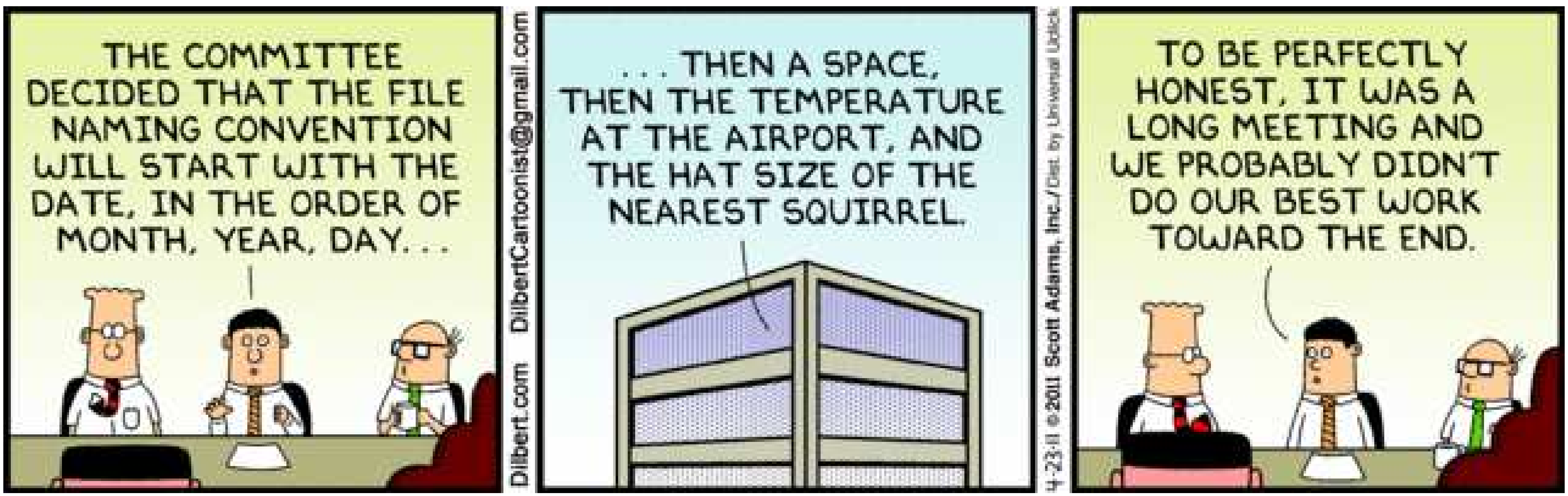
- **Displacement D(R):** `Mem[Reg[R]+D]`
  - Register R specifies start of memory region.
  - Constant displacement D specifies offset

  `mrmovq 8(%rcb),%rdx`

Conventions

It's important to understand how individual operations update the system state. *But that's not enough!*

Much of the way the Y86/x86 operates is based on a a set of *programming conventions.* Without them, you won't understand how programs work, what the compiler generates, or how your code can interact with code written by others.



Conventions

The following are conventions necessary to make programs interact:

- How do you pass arguments to a procedure?
- Where are variables (local, global, static) created?
- How does a procedure return a value?
- How do procedures preserve the state/data of the caller?

Some of these (e.g., the direction the stack grows) are reflected in specific machine operations; others are purely conventions.

Sample Program

Let's write a fragment of Y86 assembly code. Our program swaps the 8-byte values starting in memory locations 0x0100 (value A) and 0x0200 (value B).

```
start:
    xorq     %rax, %rax
    mrmovq   0x100(%rax), %rbx
    mrmovq   0x200(%rax), %rcx
    rmmovq   %rcx, 0x100(%rax)
    rmmovq   %rbx, 0x200(%rax)
    halt
```

Reg.	Use
%rax	0
%rbx	A
%rcx	B

It's usually a good idea to have a table like this to keep track of the use of registers.

Sample Program: Machine Code

Now, we generate the machine code for our sample program. Assume that it is stored in memory starting at location 0x030. *I did this by hand, so check for errors!*

```
0x030:  6300                    # xorq %rax, %rax
0x032:  50300001000000000000    # mrmovq 0x100(%rax), %rbx
0x03c:  50100002000000000000    # mrmovq 0x200(%rax), %rcx
0x046:  40100001000000000000    # rmmovq %rcx, 0x100(%rax)
0x050:  40300002000000000000    # rmmovq %rbx, 0x200(%rax)
0x05a:  00                      # halt
```

Reg.	Use
%rax	0
%rbx	A
%rcx	B

A Peek Ahead: Argument Passing

Registers: First 6 arguments

1. `%rdi`
2. `%rsi`
3. `%rdx`
4. `%rcx`
5. `%r8`
6. `%r9`

This convention is for GNU/Linux; Windows is different. Mnemonic to recall order: "Diane's silk dress cost $89."

Return value

`%rax`

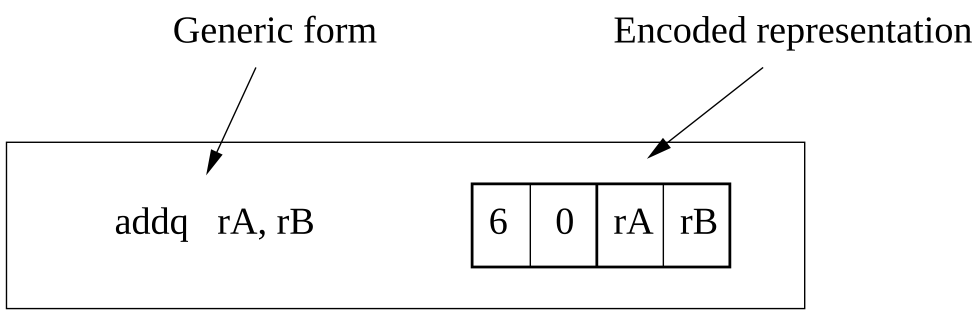
Stack: arguments 7+

...	
Arg n	
...	
Arg 8	
Arg 7	← `%rsp`

Push in reverse order. Only allocate stack space when needed.

Instruction Example

Addition Instruction



- Add value in register rA to that in register rB.
  - Store result in register rB
  - Note that Y86 only allows addition to be applied to register data.
- E.g., `addq %rax, %rsi` is encoded as: 60 06. Why?
- Set condition codes based on the result.
- Two byte encoding:
  - First indicates instruction type.
  - Second gives source and destination registers.

What effects does addq have on the state?

Effects on the State

You completely characterize an operation by saying how it changes the state.

What effects does `addq %rsi, %rdi` have on the state?

Effects on the State

You completely characterize an operation by saying how it changes the state.

What effects does `addq %rsi, %rdi` have on the state?

1. Set contents of `%rdi` to the sum of the current contents of `%rsi` and `%rdi`.
2. Set condition codes based on the result of the sum.
   - OF: set (i.e., is 1) iff the result causes an overflow
   - ZF: set iff the result is zero
   - SF: set iff the result is negative
3. Increment the program counter by 2. Why 2?

There is no effect on the memory or status flag.

Arithmetic and Logical Operations

Add

`addq rA, rB`	6	0	rA	rB

Subtract (rA from rB)

`subq rA, rB`	6	1	rA	rB

And

`andq rA, rB`	6	2	rA	rB

Exclusive Or

`xorq rA, rB`	6	3	rA	rB

- Refer to generically as "OPq"
- Encodings differ only by "function code": lower-order 4-bits in first instruction byte.
- Set condition codes as side effect.

Move Operations

Register to Register

`rrmovq rA, rB`	2	0	rA	rB

Immediate to Register

`irmovq V, rB`	3	0	F	rB	V

Register to Memory

`rmmovq rA, D(rB)`	4	0	rA	rB	D

Memory to Register

`mrmovq D(rB), rA`	5	0	rA	rB	D

- Similar to the x86-64 movq instruction.
- Similar format for memory addresses.
- Slightly different names to distinguish them.

Move Instruction Examples

x86-64	Y86	Y86 Encoding
`movq $0xabcd, %rdx`	`irmovq $0xabcd, %rdx`	30 F2 cd ab 00 00 00 00 00 0
`movq %rsp, %rbx`	`rrmovq %rsp, %rbx`	20 43
`movq -12(%rbp), %rcx`	`mrmovq -12(%rbp), %rcx`	50 15 f4 ff ff ff ff ff ff ff
`movq %rsi, 0x41c(%rsp)`	`rmmovq %rsi, 0x41c(%rsp)`	40 64 1c 04 00 00 00 00 00 0
`movq %0xabcd, (%rax)`	none	
`movq %rax, 12(%rax, %rdx)`	none	
`movq (%rbp, %rdx, 4), %rcx`	none	

The Y86 adds special move instructions to compensate for the lack of certain *addressing modes*.

Conditional Move Instructions

Move (conditionally)

`cmovXX rA, rB`	2	fn	rA	rB

- Refer to generically as “cmovXX”
- Encodings differ only by function code `fn`
- `rrmovq` instruction is a special case
- Based on values of condition codes
- Conditionally copy value from source to destination register

Note that `rrmovq` is a special case of `cmovXX`.

Conditional Move Instructions

Move Unconditionally

`rrmovq rA, rB`	2	0	rA	rB

Move when less or equal

`cmovle rA, rB`	2	1	rA	rB

Move when less

`cmovl rA, rB`	2	2	rA	rB

Move when equal

`cmove rA, rB`	2	3	rA	rB

Move when not equal

`cmovne rA, rB`	2	4	rA	rB

Move when greater or equal

`cmovge rA, rB`	2	5	rA	rB

Move when greater

`cmovg rA, rB`	2	6	rA	rB

Example of CMOV

Suppose you want to compile the following C code:

```c
long min (long x, long y) {
  if (x <= y)
    return x;
  else
    return y;
}
```

The following is one potential implementation of this. Notice that there are no jumps.

```
min:
  rrmovq %rdi, %rax            # ans <-- x
  rrmovq %rdi, %r8             # temp <-- x
  subq   %rsi, %r8             # if (temp - y) > 0
  cmovg  %rsi, %rax            #    ans <-- y
  ret                          # return ans
```

Jump Instructions

Jump (conditionally)

`jXX Dest`	7	fn	Dest

- Refer to generically as "jXX"
- Encodings differ only by function code `fn`
- Based on values of condition codes
- Same as x86-64 counterparts
- Encode full destination address (unlike PC-relative addressing in x86-64)

Jump Instructions

Jump Unconditionally

`jmp Dest`	7	0	Dest

Jump when less or equal

`jle Dest`	7	1	Dest

Jump when less

`jl Dest`	7	2	Dest

Jump when equal

`je Dest`	7	3	Dest

Jump when not equal

`jne Dest`	7	4	Dest

Jump when greater or equal

`jge Dest`	7	5	Dest

Jump when greater

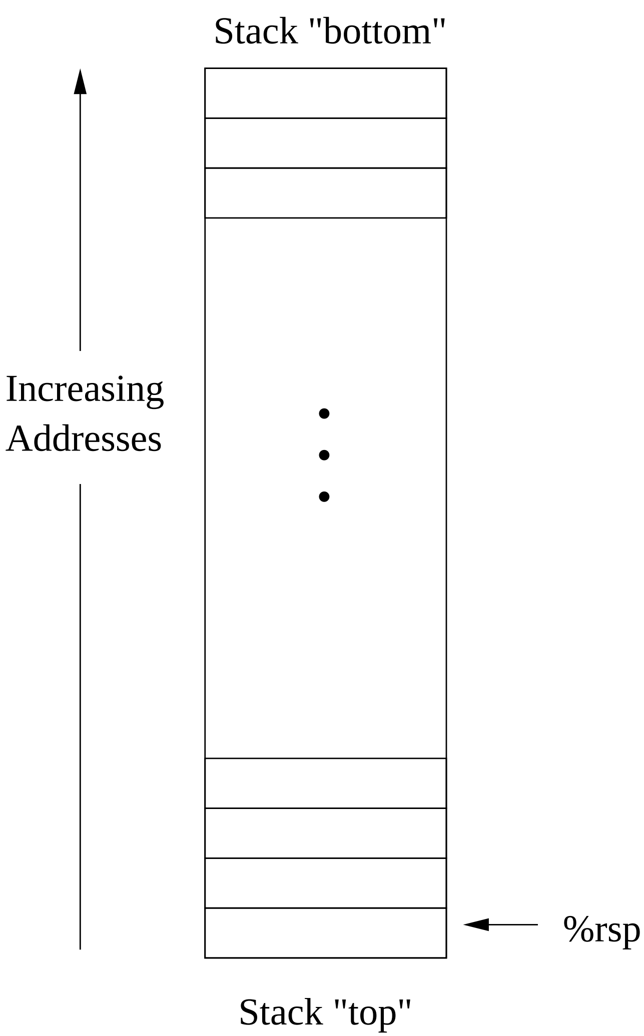
`jg Dest`	7	6	Dest

Jump Example

Suppose you want to count the number of elements in a null terminated list A with starting address in `%rdi`.

```
len:
     irmovq   $0, %rax         # result = 0
     mrmovq   (%rdi), %rdx     # val = *A
     andq     %rdx, %rdx       # Test val
     je       Done             # If 0, goto
                               #   Done
Loop:
     ....
Done:
     ret
```

Y86 Program Stack



- Region of memory holding program data.
- Used in Y86 (and x86-64) for supporting procedure calls.
- Stack top is indicated by `%rsp` , address of top stack element.
- Stack grows toward lower addresses.
  - Top element is at lowest address in the stack.
  - When pushing, must first decrement stack pointer.
  - When popping, increment stack pointer.

Stack Operations

Push

`pushq rA`	a	0	rA	F

- Decrement `%rsp` by 8.
- Store quad word from rA to memory at `%rsp` .
- Similar to x86-64 pushq operation.

Pop

`popq rA`	b	0	rA	F

- Read quad word from memory at `%rsp`.
- Save in rA.
- Increment `%rsp` by 8.
- Similar to x86-64 popq operation.

Subroutine Call and Return

Subroutine call

`call Dest`	8	0	Dest

- Push address of next instruction onto stack.
- Start executing instructions at Dest.
- Similar to x86-64 call instruction.

Subroutine return

`ret`	9	0

- Pop value from stack.
- Use as address for next instruction.
- Similar to x86-64 ret instruction.

Note that call and ret don't implement parameter/return passing. You have to do that in your code.

Miscellaneous Instructions

No operation

`nop`	1	0

- Don't do anything but advance PC.

Halt execution

`halt`	0	0

- Stop executing instructions; set status to HLT.
- x86-64 has a comparable instruction, but you can't execute it in user mode.
- We will use it to stop the simulator.
- Encoding ensures that program hitting memory initialized to zero will halt.

Status Conditions

Mnemonic	Code	Meaning
AOK	1	Normal operation
HLT	2	Halt inst. encountered
ADR	3	Bad address (instr. or data)
INS	4	Invalid instruction

Desired behavior:

- If AOK, keep executing
- Otherwise, stop program execution

Writing Y86 Code

Try to use the C compiler as much as possible.

- Write code in C.
- Compile for x86-64 with `gcc -Og -S`.
- Transliterate into Y86 code.
- Modern compilers make this more difficult, because they optimize by default.

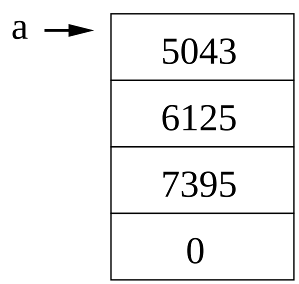
To understand Y86 (or x86) code, you have to know the meaning of the statement, but also certain *programming conventions*, especially the *stack discipline*.

- How do you pass arguments to a procedure?
- Where are local variables created?
- How does a procedure return a value?
- How do procedures save and restore the state of the caller?

Writing Y86 Code: Example

Coding example: Find number of elements in a null-terminated list.

```
long len( long a[] );
```



The answer in this case should be 3.

Y86-64 Code Generation Example

First try writing typical array code:

```
/* Count elements in null-
   terminated list */
long len( long a[] )
{
   long length;
   for (length = 0; a[
      length]; length++ );
   return length;
}
```

Compile with `gcc -Og -S`

Problem: Hard to do array indexing on Y86, since we don't have scaled addressing modes.

x86 Code:

```
L3:
    addq   $1, %rax
    cmpq   $0, (%rdi,%rax,8)
    jne    L3
```

Y86-64 Code Generation Example (2)

Second try: Write C code that mimics expected Y86 code.

```
/* Count elements in null-
   terminated list */
long len2( long *a )
{
    long ip = (long) a;
    long val = *(long *) ip;
    long len = 0;
    while (val) {
        ip += sizeof(long);
        len++;
        val = *(long *)  ip;
    }
    return len;
}
```

Result:

- Compiler generates exact same code as before!
- Compiler converts both versions into the same intermediate form.

Y86-64 Code Generation Example (3)

```
len:
        irmovq   $1, %r8          # Constant 1
        irmovq   $8, %r9          # Constant 8
        irmovq   $0, %rax         # len = 0
        mrmovq   (%rdi), %rdx     # val = *a
        andq     %rdx, %rdx       # Test val
        je       Done             # If 0, goto
                                  #   Done
Loop:
        addq     %r8, %rax        # len++
        addq     %r9, %rdi        # a++
        mrmovq   (%rdi), %rdx     # val = *a
        andq     %rdx, %rdx       # Test val
        jne      Loop             # If !0, goto
                                  #   Loop
Done:
        ret
```

Reg.	Use
%rdi	a
%rax	len
%rdx	val
%r8	1
%r9	8

Y86 Sample Program Structure

```
init:            # Initialization
   ...
   call Main
   halt

   .align 8      # Program data
Array:
   ...

Main:            # Main function
   ...
   call len
   ...

len:             # Length function
   ...

   .pos 0x100    # Place stack
Stack:
```

- Program starts at address 0
- Must set up stack
  - Where located
  - Pointer values
  - Mustn't overwrite data
- Must initialize data

Y86 Program Structure (2)

```
init:
      # Set up stack pointer
      irmovq  Stack, %rsp
      # Execute main program
      call Main
      # Terminate
      halt

# Array of 4 elements + final 0
      .align 8
Array:
      .quad 0x000d000d000d000d
      .quad 0x00c000c000c000c0
      .quad 0x0b000b000b000b00
      .quad 0xa000a000a000a000
      .quad 0
```

- Program starts at address 0
- Must set up stack
- Must initialize data
- Can use symbolic names

Y86 Program Structure (3)

```
Main:
      irmovq  Array, %rdi
      # call len(Array)
      call    len
      ret
```

Set up call to len:

- Follow x86-64 procedure conventions
- Pass array address as argument

Y86 Assembler

A program that translates Y86 code into machine language.

- 1-1 mapping of instructions to encodings.
- Resolves symbolic names.
- Translation is linear.
- Assembler directives give additional control.

Some common directives:

- `.pos x`: subsequent lines of code start at address x.
- `.align x`: align the next line to an x-byte boundary (e.g., long ints should be at a quadword address, divisible by 8).
- `.quad x`: put an 8-byte value x at the current address; a way to initialize a value.

Assembling Y86 Program

```
unix> yas len.ys
```

- Generates "object code" file `len.yo`
- Actually looks like disassembler output

```
0x054:                        | len:
0x054: 30f80100000000000000   |   irmovq  $1, %r8
0x05e: 30f90800000000000000   |   irmovq  $8, %r9
0x068: 30f00000000000000000   |   irmovq  $0, %rax
0x072: 50270000000000000000   |   mrmovq  (%rdi), %rdx
0x07c: 6222                   |   andq    %rdx, %rdx
0x07e: 73a000000000000000     |   je      Done
0x087:                        | Loop:
0x087: 6080                   |   addq    %r8, %rax
0x089: 6097                   |   addq    %r9, %rdi
0x08b: 50270000000000000000   |   mrmovq  (%rdi), %rdx
0x095: 6222                   |   andq    %rdx, %rdx
0x097: 748700000000000000     |   jne     Loop
0x0a0:                        | Done:
0x0a0: 90                     |   ret
```

Simulating Y86 Programs

```
unix>  yis len.yo
```

Instruction set simulator

- Computes effect of each instruction on process state
- Prints changes in state from original

```
Stopped in 33 steps at PC = 0x13, Status 'HLT', CC Z=1
    S=0 O=0
Changes to registers:
%rax:   0x0000000000000000    0x0000000000000004
%rsp:   0x0000000000000000    0x0000000000000100
%rdi:   0x0000000000000000    0x0000000000000038
%r8:    0x0000000000000000    0x0000000000000001
%r9:    0x0000000000000000    0x0000000000000008

Changes to memory:
0x00f0: 0x0000000000000000    0x0000000000000053
0x00f8: 0x0000000000000000    0x0000000000000013
```

CISC Instruction Sets

Complex Instruction Set Computer

- Dominant ISA style through the 80s.
- Lots of instructions:
  - Variable length
  - Stack as mechanism for supporting functions
  - Explicit push and pop instructions.
- ALU instructions can access memory.
  - E.g., `addq %rax, 12(%rbx, %rcx, 8)`
  - Requires memory read and write in one instruction execution.
  - Some ISAs had much more complex address calculations.
- Set condition codes as a side effect of other instructions.
- Basic philosophy:
  - Memory is expensive;
  - Instructions to support high-level language constructs.

RISC Instruction Sets

Reduced Instruction Set Computer

- Originated in IBM Research; popularized in Berkeley and Stanford projects.
- Few, simple instructions.
  - Takes more instructions to execute a task, but faster and simpler implementation
  - Fixed length instructions for simpler decoding
- Register-oriented ISA
  - More registers (32 typically)
  - Stack is back-up for registers
- Only load and store instructions can access memory (mrmovq and rmmovq in Y86).
- Explicit test instructions set condition values in register.
- Philosophy: KISS

CISC vs. RISC

Original Debate

- Strong opinions!
- CISC proponents–easy for compiler, fewer code bytes
- RISC proponents–better for optimizing compilers, can make run fast with simple chip design

Current Status

- For desktop processors, choice of ISA not a technical issue
  - With enough hardware, can make anything run fast
  - Code compatibility more important
- x86-64 adopted many RISC features
  - More registers; use them for argument passing
- For embedded processors, RISC makes sense
  - Smaller, cheaper, less power
  - Most cell phones use ARM processor

Summary

Y86-64 Instruction Set Architecture

- Similar state and instructions to x86-64
- Simpler encodings
- Somewhere between CISC and RISC

How Important is ISA Design?

- Less now than before: with enough hardware, can make almost anything run fast!