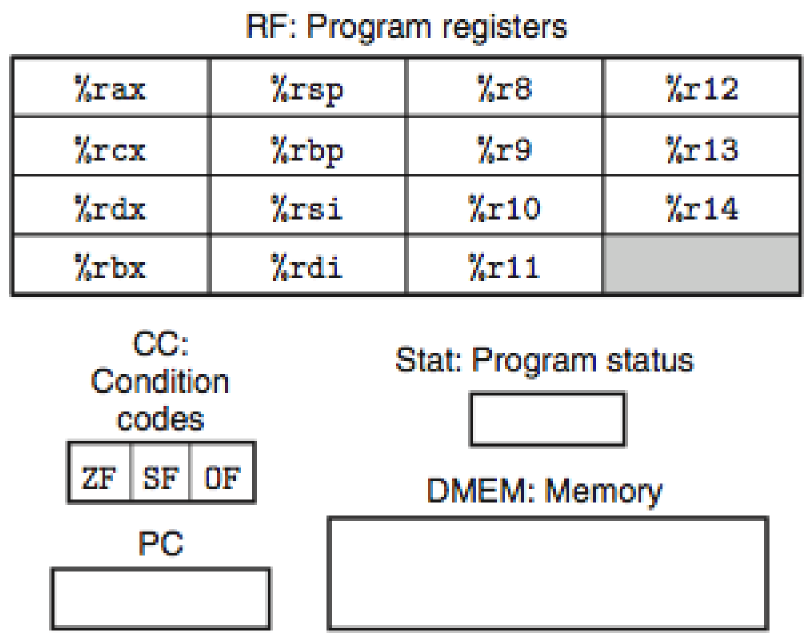
COM 410 – Slides 2
The X86-64 Processor

Y86-64 ISA
(Simplified X86-64 ISA)
ISA = Instruction Set Architecture

Understanding how the processor works lets you understand how the overall computer system works

Helps make decisions about how to design hardware systems

Someday, you might actually work on processor design



PROGRAM STATE: each instruction in a Y86-64 program can read and modify some part of the processor state. This is referred to as the programmer-visible state, where the "programmer" in this case is either someone writing programs in assembly code or a compiler generating machine-level code.

There are 15 program registers:

%rax, %rcx, %rdx, %rbx, %rsp, %rbp, %rsi, %rdi, and %r8 through %r14.

(x86-64 register %r15 ommitted to simplify the instruction encoding.)

Each of these stores a 64-bit word. Register %rsp is used as a stack pointer by the push, pop, call, and return instructions.

There are three single-bit condition codes, ZF, SF, and OF, storing information about the effect of the most recent arithmetic or logical instruction.

The program counter (PC) holds the address of the instruction currently being executed.

The memory is conceptually a large array of bytes, holding both program and data. A combination of hardware and operating system software translates these into the actual, or physical, addresses indicating where the values are actually stored in memory.

A final part of the program state is a status code Stat, indicating the overall state of program execution. It will indicate either normal operation or that some sort of exception has occurred, such as when an instruction attempts to read from an invalid memory address

```
main:
  subq    $8, %rsp
  movl    $.LC0, %edi
  call    puts
  movl    $0, %eax
  addq    $8, %rsp
  ret
```

Assembly language version of ‘hello.c’

Byte	0	1	2–9
`halt`	0 0		
`nop`	1 0		
`rrmovq` **rA, rB**	2 0	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 (bytes 1–8)	
`cmovXX` **rA, rB**	2 fn	rA rB	
`call` **Dest**	8 0	Dest (bytes 1–8)	
`ret`	9 0		
`pushq` **rA**	A 0	rA F	
`popq` **rA**	B 0	rA F	

The x86-64 movq instruction is split into four different instructions: irmovq, rrmovq, mrmovq, and rmmovq, explicitly indicating the form of the source and destination.

The source is either immediate (i), register (r), or memory (m). It is designated by the first character in the instruction name. The destination is either register (r) or memory (m).

There are four integer operation instructions Opq included in the figure: addq, subq, andq, and xorq.

They operate only on register data, whereas x86-64 also allows operations on memory data.

These instructions set the three condition codes ZF, SF, and OF (zero, sign, and overflow).

The seven jump instructions jXX are jmp, jle, jl, je, jne, jge, and jg.

There are six conditional move instructions cmovXX: cmovle, cmovl, cmove, cmovne, cmovge, and cmovg. These have the same format as the register–register move instruction rrmovq, but the destination register is updated only if the condition codes satisfy the required constraints.

The call instruction pushes the return address on the stack and jumps to the destination address. The ret instruction returns from such a call.

The pushq and popq instructions implement push and pop, just as they do in x86-64.

The halt instruction stops instruction execution. x86-64 has a comparable instruction, called hlt.

```
long sum(long *start, long count)
{
    long sum = 0;
    while (count) {
        sum += *start;
        start++;
        count--;
    }
    return sum;
}
```

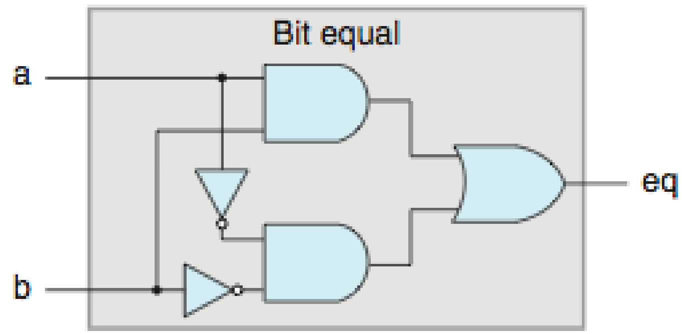
X86-64 CODE

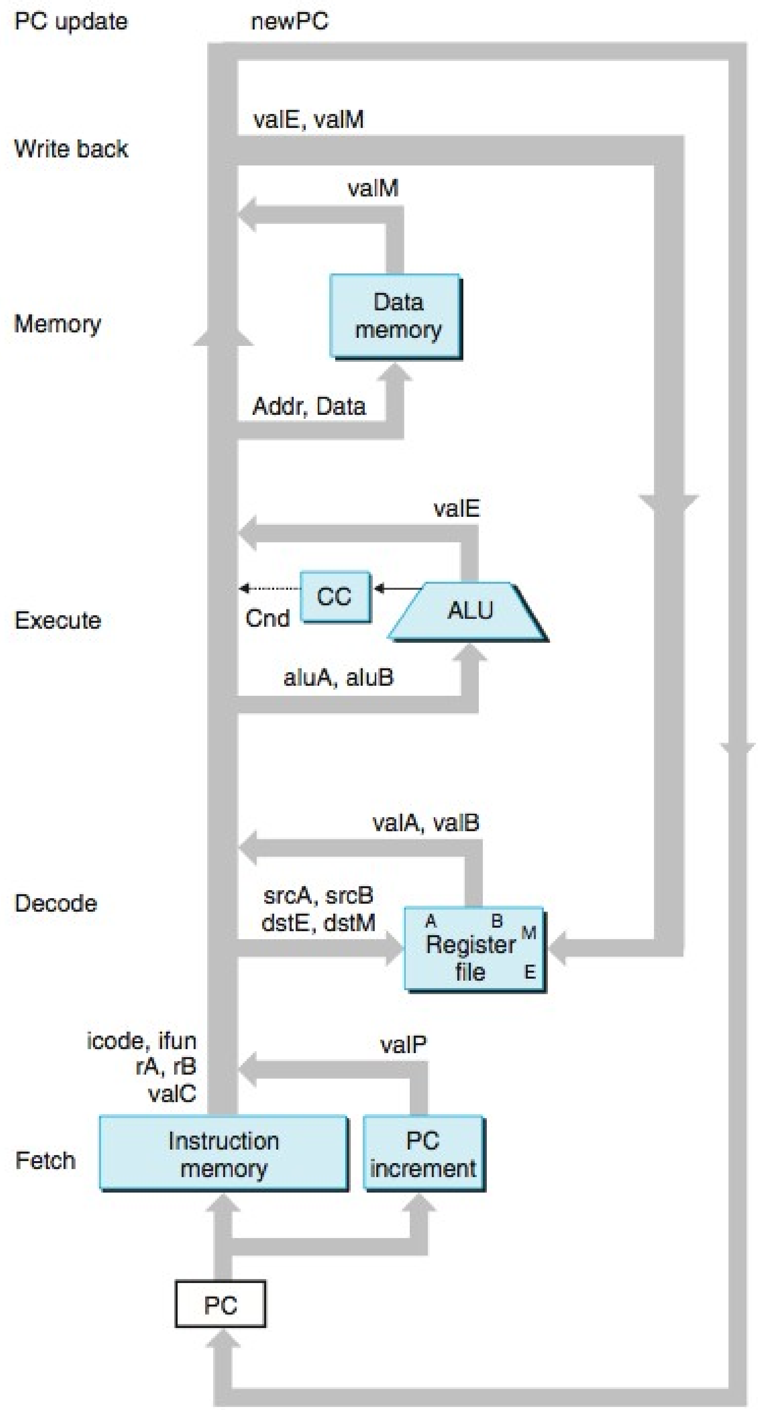
```
sum:
  movl    $0, %eax          sum = 0
  jmp     .L2               Goto test
.L3:                      loop:
  addq    (%rdi), %rax      Add *start to sum
  addq    $8, %rdi          start++
  subq    $1, %rsi          count--
.L2:                      test:
  testq   %rsi, %rsi        Test sum
  jne     .L3               If !=0, goto loop
  rep; ret                  Return
```

Y86-64 CODE

```
sum:
    irmovq $8,%r8           Constant 8
    irmovq $1,%r9           Constant 1
    xorq %rax,%rax          sum = 0
    andq %rsi,%rsi          Set CC
    jmp    test             Goto test
loop:
    mrmovq (%rdi),%r10      Get *start
    addq %r10,%rax          Add to sum
    addq %r8,%rdi           start++
    subq %r9,%rsi           count--.  Set CC
test:
    jne    loop             Stop when 0
    ret                     Return
```

```
# Execution begins at address 0
        .pos 0
        irmovq stack, %rsp      # Set up stack pointer
        call main               # Execute main program
        halt                    # Terminate program

# Array of 4 elements
        .align 8
array:
        .quad 0x000d000d000d
        .quad 0x00c000c000c0
        .quad 0x0b000b000b00
        .quad 0xa000a000a000

main:
        irmovq array,%rdi
        irmovq $4,%rsi
        call sum                # sum(array, 4)
        ret

# long sum(long *start, long count)
# start in %rdi, count in %rsi
sum:
        irmovq $8,%r8        # Constant 8
        irmovq $1,%r9        # Constant 1
        xorq %rax,%rax       # sum = 0
        andq %rsi,%rsi       # Set CC
        jmp     test         # Goto test
loop:
        mrmovq (%rdi),%r10   # Get *start
        addq %r10,%rax       # Add to sum
        addq %r8,%rdi        # start++
        subq %r9,%rsi        # count--.  Set CC
test:
        jne    loop          # Stop when 0
        ret                  # Return

# Stack starts here and grows to lower addresses
        .pos 0x200
stack:
```


Bit equal
a
b
eq



PC update
newPC
Write back
valE, valM
valM
Memory
Data memory
Addr, Data
valE
Execute
CC
ALU
Cnd
aluA, aluB
valA, valB
Decode
srcA, srcB
dstE, dstM
A
B
M
Register file
E
icode, ifun
rA, rB
valC
valP
Fetch
Instruction memory
PC increment
PC

TOPICS FROM CHAPTERS 2-3

- Hexadecimal Notation
- Integer Representations
- Integer Arithmetic
- Floating Point
- Program Encodings
- Data Formats
- Accessing Information
- Arithmetic and Logical Operations
- Control
- Procedures
- Array Allocation and Access