Y86-64 ISA

'20H2

송 인 식

Outline

- Y86-64 ISA
  - Programmer-visible state
  - Instruction Format
  - Exceptions
- Y86-64 Programs
- CISC vs. RISC

Y86-64

- A strip-down version of x86-64 created by textbook authors for educational purposes
  - Similar state and instructions
  - Simpler encodings
  - Somewhere between CISC and RISC
- We will work through a CPU design example with Y86-64 in Chap. 4
- Y86-64 toolset available at http://csapp.cs.cmu.edu/3e/sim.tar
  - yas (assembler), yis (ISA simulator)
  - hcl2c (HCL to C translator), hcl2v (HCL to Verilog translator)
  - ssim, ssim+, psim (hardware simulator)

Y86-64 Programmer-Visible State

RF: Program registers

%rax	%rsp	%r8	%r12
%rcx	%rbp	%r9	%r13
%rdx	%rsi	%r10	%r14
%rbx	%rdi	%r11	

CC: Condition codes

ZF	SF	OF

Stat: Program status

PC

DMEM: Memory

Y86-64 Programmer-Visible State

- Program Registers
  - 15 registers (no %r15), each 64 bits
- Condition codes
  - Single-bit flags set by arithmetic or logical instructions
  - ZF: Zero, SF: Negative, OF: Overflow
- Program Counter: address of next instruction
- Program Status: normal vs. error condition
- Memory
  - Byte-addressable storage array
  - Words stored in little-endian byte order

Y86-64 Instruction Set #1

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								

Y86-64 Instructions

- 1 – 10 bytes of information read from memory
  - Can determine instruction length from first byte
- Only supports 64-bit operations
- RISC style
  - Not as many instruction types, and simpler encoding than with x86-64
  - Simple addressing mode: D(rA)
  - ALU instructions operate on registers (not memory)
  - Registers are specified in the fixed location, if any
- Each accesses and modifies some part(s) of the program state

Y86-64 [Conditional] Move Instructions

Byte	0		1		2–9
`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	

`rrmovq`	2	0
`cmovle`	2	1
`cmovl`	2	2
`cmove`	2	3
`cmovne`	2	4
`cmovge`	2	5
`cmovg`	2	6

Y86-64 ALU Instructions

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								

addq	6 0
subq	6 1
andq	6 2
xorq	6 3

Y86-64 [Conditional] Branch Instructions

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								

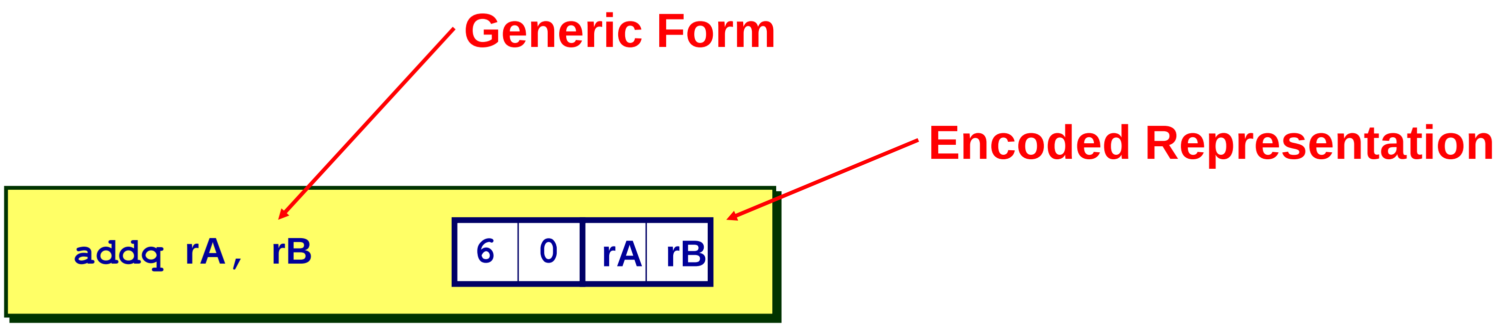
`jmp`	7 0
`jle`	7 1
`jl`	7 2
`je`	7 3
`jne`	7 4
`jge`	7 5
`jg`	7 6

Encoding Registers

- Each register has 4-bit ID
  - Same encoding as in x86-64
- Register ID 15 (0xf) indicates "no register"
  - Will use this in our hardware design in multiple places

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

Instruction Example: addq



- Add value in register rA to that in register rB
  - Store result in register rB
  - Note that Y86-64 only allows addition to be applied to register data
- Set condition codes based on result
- e.g., addq %rax,%rsi   Encoding: 60 06
- Two-byte encoding
  - First indicates instruction type
  - Second gives source and destination registers

Arithmetic and Logical Operations

Instruction Code

Function Code

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"
  - Low-order 4 bits in the first instruction byte
- Set condition codes as side effect

Move Operations

Register → Register

`rrmovq rA, rB`

2	0	rA	rB	

Immediate → Register

`irmovq V, rB`

3	0	F	rB	V

Register → Memory

`rmmovq rA, D(rB)`

4	0	rA	rB	D

Memory → Register

`mrmovq D(rB), rA`

5	0	rA	rB	D

- Like the x86-64 movq instruction
- Simpler format for memory addresses
- Give different names to keep them distinct

Move Instruction Examples

X86-64	Y86-64
`movq $0xabcd, %rdx`	`irmovq $0xabcd, %rdx`

Encoding: `30 82 cd ab 00 00 00 00 00 00`

`movq %rsp, %rbx`	`rrmovq %rsp, %rbx`

Encoding: `20 43`

`movq -12(%rbp),%rcx`	`mrmovq -12(%rbp),%rcx`

Encoding: `50 15 f4 ff ff ff ff ff ff ff`

`movq %rsi,0x41c(%rsp)`	`rmmovq %rsi,0x41c(%rsp)`

Encoding: `40 64 1c 04 00 00 00 00 00 00`

Conditional Move Instructions

Move Unconditionally

Instruction				
`rrmovq rA, rB`	2	0	rA	rB

Move When Less or Equal

Instruction				
`cmovle rA, rB`	2	1	rA	rB

Move When Less

Instruction				
`cmovl rA, rB`	2	2	rA	rB

Move When Equal

Instruction				
`cmove rA, rB`	2	3	rA	rB

Move When Not Equal

Instruction				
`cmovne rA, rB`	2	4	rA	rB

Move When Greater or Equal

Instruction				
`cmovge rA, rB`	2	5	rA	rB

Move When Greater

Instruction				
`cmovg rA, rB`	2	6	rA	rB

- Refer to generically as "`cmovXX`"
- Encodings differ only by "function code"
- Based on values of condition codes
- Variants of `rrmovq` instruction
  - (Conditionally) copy value from source to destination register

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

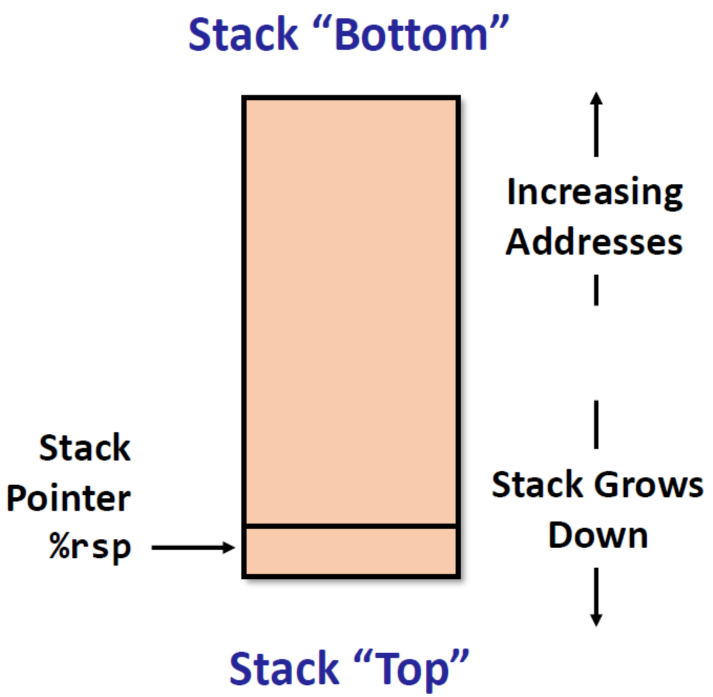
Jump When Greater

`jg` Dest	7	6	Dest

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

Y86 Program Stack

- Same as in x86-64
  - Region of memory holding program data
  - Used for supporting procedure calls
- Stack top 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
  - After popping, increment stack pointer



Stack Operations

- **pushq**
  - Decrement %rsp by 8
  - Store word from rA to memory at %rsp
  - Like x86-64

pushq rA	A	0	rA	F

- **popq**
  - Read word from memory at %rsp
  - Save in rA
  - Increment %rsp by 8
  - Like x86-64

popq rA	B	0	rA	F

Subroutine Call and Return

- **call**
  - Push address of next instruction onto stack
  - Start executing instructions at Dest
  - Like x86-64

`call` **Dest**	8	0	Dest

- **ret**
  - Pop value from stack
  - Use as address for next instruction
  - Like x86-64

`ret`	9	0

Miscellaneous Instructions

- **nop**
  - Don’t do anything

- **halt**
  - Stop executing instructions
  - x86-64 has comparable instruction, but 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

- AOK
  - Normal operation
- HLT
  - Halt instruction encountered
- ADR
  - Bad address (either instruction or data) encountered
- INS
  - Invalid instruction encountered

- If AOK, keep going. Otherwise, stop program execution

Mnemonic	Code
AOK	1

Mnemonic	Code
HLT	2

Mnemonic	Code
ADR	3

Mnemonic	Code
INS	4

Outline

- Y86-64 ISA
  - Programmer-visible state
  - Instruction Format
  - Exceptions
- Y86-64 Programs
- CISC vs. RISC

Example: max.ys

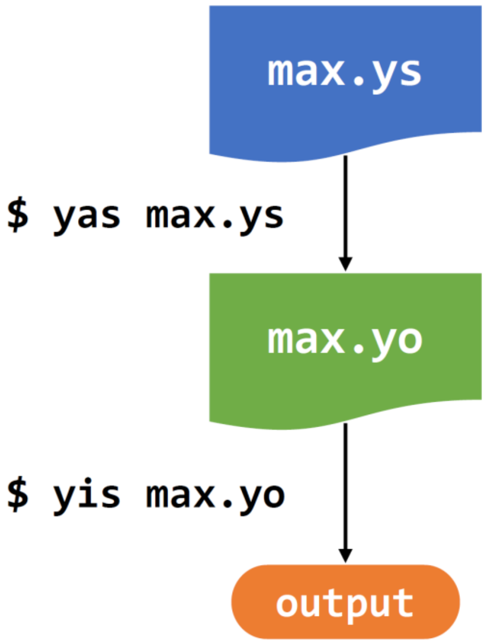
```
# Execution begins at address 0
    .pos 0
    irmovq stack, %rsp          # Set up stack pointer
    call main                   # Call main
    halt                        # Terminate program

# Global data
    .align 8
array:
    .quad 0x0000000000000003
    .quad 0x0000000000000004
    .quad 0x00000b000b000b00

main:
    irmovq array, %rbx          # %rbx <- array
    mrmovq (%rbx), %rdi         # %rdi <- array[0]
    mrmovq 8(%rbx), %rsi        # %rsi <- array[1]
    call max                    # Call max
    rmmovq %rax, 16(%rbx)       # array[2] <- %rax
    ret

# long max(long x, y)
max:
    rrmovq %rdi, %rax           # %rax <- x
    subq %rsi, %rdi             # %rdi <- x - y; set flag
    cmovl %rsi, %rax            # if (x < y) %rax <- y
    ret

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

Example: max.yo

- `yas max.ys`

```
                              | # Execution begins at address 0
0x000:                        |     .pos 0
0x000: 30f40002000000000000   |     irmovq stack, %rsp # Set up stack pointer
0x00a: 803000000000000000     |     call main # Call main
0x013: 00                     |     halt # Terminate program
                              |
                              | # Global data
0x018:                        |     .align 8
0x018:                        | array:
0x018: 0300000000000000       |     .quad 0x0000000000000003
0x020: 0400000000000000       |     .quad 0x0000000000000004
0x028: 000b000b000b0000       |     .quad 0x00000b000b000b00
                              |
0x030:                        | main:
0x030: 30f31800000000000000   |     irmovq array, %rbx # %rbx <- array
0x03a: 50730000000000000000   |     mrmovq (%rbx), %rdi # %rdi <- array[0]
0x044: 50630800000000000000   |     mrmovq 8(%rbx), %rsi # %rsi <- array[1]
0x04e: 806200000000000000     |     call max # Call max
0x057: 40031000000000000000   |     rmmovq %rax, 16(%rbx) # array[2] <- %rax
0x061: 90                     |     ret
                              |
                              | # long max(long x, y)
0x062:                        | max:
0x062: 2070                   |     rrmovq %rdi, %rax # %rax <- x
0x064: 6167                   |     subq %rsi, %rdi # %rdi <- x - y; set flag
0x066: 2260                   |     cmovl %rsi, %rax # if (x < y) %rax <- y
0x068: 90                     |     ret
                              |
                              | # The stack starts here and grows to lower addresses
0x200:                        | .pos 0x200
0x200:                        | stack:
```

Example: Running max.yo on yis




```
$ yis max.yo
Stopped in 13 steps at PC = 0x13. Status 'HLT', CC Z=0 S=1
O=0
Changes to registers:
%rax: 0x0000000000000000 0x0000000000000004
%rbx: 0x0000000000000000 0x0000000000000018
%rsp: 0x0000000000000000 0x0000000000000200
%rsi: 0x0000000000000000 0x0000000000000004
%rdi: 0x0000000000000000 0xffffffffffffffff

Changes to memory:
0x0028: 0x00000b000b000b00 0x0000000000000004
0x01f0: 0x0000000000000000 0x0000000000000057
0x01f8: 0x0000000000000000 0x0000000000000013
```

Sample C Program

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

x86-64 Code

```
long sum(long *start, long count)
start in %rdi, count in %rsi
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

```
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
```

Full Y86-64 Code

```
# 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:
```

Output of YAS Assembler

```
                              | # Execution begins at address 0
0x000:                        |   .pos 0
0x000: 30f40002000000000000 |   irmovq stack, %rsp      # Set up stack pointer
0x00a: 803800000000000000   |   call main               # Execute main program
0x013: 00                     |   halt                    # Terminate program
                              |
                              | # Array of 4 elements
0x018:                        |   .align 8
0x018:                        | array:
0x018: 0d000d000d000000       |   .quad 0x000d000d000d
0x020: c000c000c0000000       |   .quad 0x00c000c000c0
0x028: 000b000b000b0000       |   .quad 0x0b000b000b00
0x030: 00a000a000a00000       |   .quad 0xa000a000a000
                              |
0x038:                        | main:
0x038: 30f71800000000000000 |   irmovq array,%rdi
0x042: 30f60400000000000000 |   irmovq $4,%rsi
0x04c: 805600000000000000   |   call sum                # sum(array, 4)
0x055: 90                     |   ret
```

Output of YAS Assembler

```
                              | # long sum(long *start, long count)
                              | # start in %rdi, count in %rsi
0x056:                        | sum:
0x056: 30f80800000000000000   |   irmovq $8,%r8        # Constant 8
0x060: 30f90100000000000000   |   irmovq $1,%r9        # Constant 1
0x06a: 6300                   |   xorq %rax,%rax       # sum = 0
0x06c: 6266                   |   andq %rsi,%rsi       # Set CC
0x06e: 708700000000000000     |   jmp    test          # Goto test
0x077:                        | loop:
0x077: 50a70000000000000000   |   mrmovq (%rdi),%r10   # Get *start
0x081: 60a0                   |   addq %r10,%rax       # Add to sum
0x083: 6087                   |   addq %r8,%rdi        # start++
0x085: 6196                   |   subq %r9,%rsi        # count--.  Set CC
0x087:                        | test:
0x087: 747700000000000000     |   jne    loop          # Stop when 0
0x090: 90                     |   ret                  # Return
                              |
                              | # Stack starts here and grows to lower addresses
0x200:                        |   .pos 0x200
0x200:                        | stack:
```

Output of YIS Simulator

```
Stopped in 34 steps at PC = 0x13.  Status 'HLT', CC Z=1 S=0 O=0
Changes to registers:
%rax:   0x0000000000000000      0x0000abcdabcdabcd
%rsp:   0x0000000000000000      0x0000000000000200
%rdi:   0x0000000000000000      0x0000000000000038
%r8:    0x0000000000000000      0x0000000000000008
%r9:    0x0000000000000000      0x0000000000000001
%r10:   0x0000000000000000      0x0000a000a000a000

Changes to memory:
0x01f0: 0x0000000000000000      0x0000000000000055
0x01f8: 0x0000000000000000      0x0000000000000013
```

Outline

- Y86-64 ISA
  - Programmer-visible state
  - Instruction Format
  - Exceptions
- Y86-64 Programs
- CISC vs. RISC

CISC (Complex Instruction Set Computer)

- Stack-oriented instruction set
  - Use stack to pass arguments, save program counter
  - Explicit push and pop instructions
- Arithmetic instructions can access memory
  - Requires memory read and write: e.g. addq %rax, 12(%rbx,%rcx,8)
  - Complex address calculation
- Condition codes
  - Set as side effect of arithmetic and logical instructions
- Philosophy
  - Add instructions to perform “typical” programming tasks
- DEC PDP-11 & VAX, IBM System/360, Motorola 68000, IA32, x86-64, … (Dominant style through mid-80’s)

RISC (Reduced Instruction Set Computer)

- Fewer, simpler instructions
  - Might take more to get given task done
  - Can execute them with small and fast hardware
  - Stanford MIPS, UCB RISC, Sun SPARC, IBM Power/PowerPC, ARM, SuperH, …
- Register-oriented instruction set
  - Many more (typically 32+) registers
  - Use for arguments, return pointer, temporaries
- Only load and store instructions can access memory
  - Similar to Y86-64 mrmovq and rmmovq
- No condition codes
  - Test instructions return 0/1 in register

MIPS Registers

Register	Name	Use
$0	$0	Constant 0
$1	$at	Reserved Temp.
$2	$v0	Return Values
$3	$v1	
$4	$a0	Procedure arguments
$5	$a1	
$6	$a2	
$7	$a3	
$8	$t0	Caller Save Temporaries: May be overwritten by called procedures
$9	$t1	
$10	$t2	
$11	$t3	
$12	$t4	
$13	$t5	
$14	$t6	
$15	$t7	

Register	Name	Use
$16	$s0	Callee Save Temporaries: May not be overwritten by called procedures
$17	$s1	
$18	$s2	
$19	$s3	
$20	$s4	
$21	$s5	
$22	$s6	
$23	$s7	
$24	$t8	Caller Save Temp
$25	$t9	
$26	$k0	Reserved for Operating Sys
$27	$k1	
$28	$gp	Global Pointer
$29	$sp	Stack Pointer
$30	$s8	Callee Save Temp
$31	$ra	Return Address

MIPS Instruction Examples

R-R

Op	Ra	Rb	Rd	00000	Fn

```
addu $3,$2,$1          # Register add: $3 = $2+$1
```

Load/Store

Op	Ra	Rb	Offset

```
lw $3,16($2)           # Load Word: $3 = M[$2+16]

sw $3,16($2)           # Store Word: M[$2+16] = $3
```

Branch

Op	Ra	Rb	Offset

```
beq $3,$2,dest         # Branch when $3 = $2
```

Jump

Op	Dest

```
jmp   Dest             # Jump to dest
```

CISC vs. RISC

- Original debate
  - 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 (e.g. most cell phones use ARM processor)

Summary

- Y86-64 Instruction Set Architecture
  - Similar state and instructions as 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 go fast

Questions?