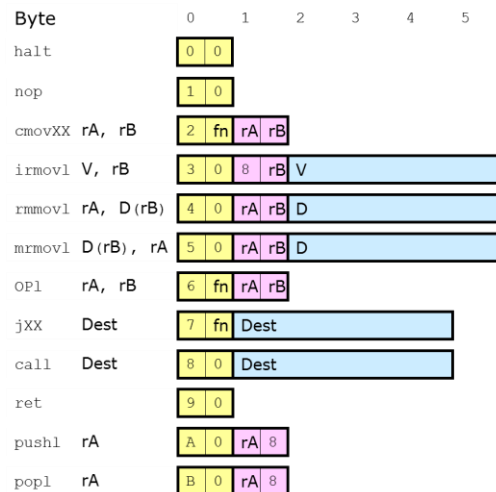


Processor Architecture

Overview and the Y86 ISA



Overview – The Y86 Processor

- A (much, much) simpler version of the X86 but powerful enough to face many of the challenges encountered by processor designers.
- Our approach of processor architecture design covers
 - the design of the Y86 ISA
 - the microarchitecture level
 - ▶ basic hardware blocks are given (ALUs, register files, memory)
 - ▶ implement control logic to make sure each instruction flows through properly
- Implementation-wise we look at
 - a sequential implementation of the Y86
 - ▶ a simple, but not very fast processor design
 - the concepts of pipelining
 - ▶ get more things running simultaneously
 - a pipelined implementation of the Y86
 - ▶ get an idea of what it takes to implement a pipelined processor

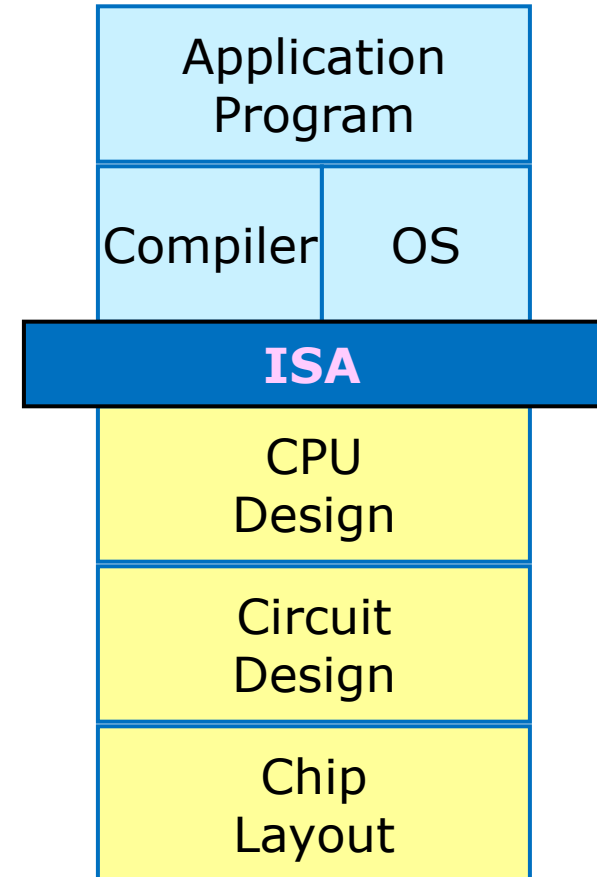
Recap – The Role of the ISA

■ Assembly Language View

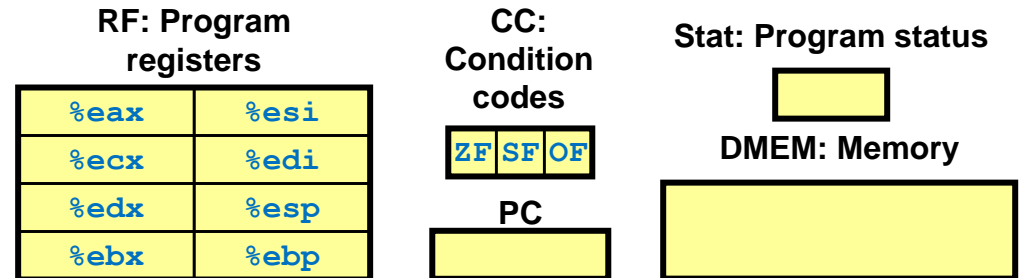
- Processor state
 - ▶ Registers, memory, ...
- Instructions
 - ▶ `addl, pushl, ret, ...`
 - ▶ How instructions are encoded as bytes

■ Layer of Abstraction

- Above: how to program machine
 - ▶ Processor executes instructions in a sequence
- Below: what needs to be built
 - ▶ Use variety of tricks to make it run fast
 - ▶ E.g., execute multiple instructions simultaneously



Y86 Processor State



■ Program Registers

- Same 8 as with IA32. Each 32 bits

■ Condition Codes

- Single-bit flags set by arithmetic or logical instructions

▶ ZF : Zero

SF : Negative

OF : Overflow

■ Program Counter

- Indicates address of next instruction

■ Program Status

- Indicates either normal operation or some error condition

■ Memory

- Byte-addressable storage array
- Words stored in little-endian byte order

Y86 Instruction Set Overview

Byte	0	1	2	3	4	5
halt	0	0				
nop	1	0				
cmovXX rA, rB	2	fn	rA	rB		
irmovl V, rB	3	0	8	rB	V	
rmmovl rA, D(rB)	4	0	rA	rB	D	
mrmovl D(rB), rA	5	0	rA	rB	D	
OpI rA, rB	6	fn	rA	rB		
jXX Dest	7	fn	Dest			
call Dest	8	0	Dest			
ret	9	0				
pushl rA	A	0	rA	8		
popl rA	B	0	rA	8		

Y86 Instructions

■ Format

- 1–6 bytes of information read from memory
 - ▶ Can determine instruction length from first byte
 - ▶ Not as many instruction types, and simpler encoding than with IA32
- Each accesses and modifies some part(s) of the program state

Y86 Instruction Set – cmovXX

Byte	0	1	2	3	4	5
halt	0	0				
nop	1	0				
cmovXX rA, rB	2	fn	rA	rB		
irmovl V, rB	3	0	8	rB	V	
rmmovl rA, D(rB)	4	0	rA	rB	D	
rrmovl D(rB), rA	5	0	rA	rB	D	
OpI rA, rB	6	fn	rA	rB		
jXX Dest	7	fn	Dest			
call Dest	8	0	Dest			
ret	9	0				
pushl rA	A	0	rA	8		
popl rA	B	0	rA	8		

rrmovl	7	0
cmovle	7	1
cmovl	7	2
cmove	7	3
cmovne	7	4
cmovge	7	5
cmovg	7	6

Y86 Instruction Set – OPl

Byte	0	1	2	3	4	5
halt	0	0				
nop	1	0				
cmovXX rA, rB	2	fn	rA	rB		
irmovl V, rB	3	0	8	rB	V	
rmmovl rA, D(rB)	4	0	rA	rB	D	
mrmmovl D(rB), rA	5	0	rA	rB	D	
OPl rA, rB	6	fn	rA	rB		
jXX Dest	7	fn	Dest			
call Dest	8	0	Dest			
ret	9	0				
pushl rA	A	0	rA	8		
popl rA	B	0	rA	8		

addl	6	0
subl	6	1
andl	6	2
xorl	6	3

Y86 Instruction Set – jXX

Byte	0	1	2	3	4	5
halt	0	0				
nop	1	0				
cmovXX rA, rB	2	fn	rA	rB		
irmovl V, rB	3	0	8	rB	V	
rmmovl rA, D(rB)	4	0	rA	rB	D	
mrmmovl D(rB), rA	5	0	rA	rB	D	
OpI rA, rB	6	fn	rA	rB		
jXX Dest	7	fn	Dest			
call Dest	8	0	Dest			
ret	9	0				
pushl rA	A	0	rA	8		
popl rA	B	0	rA	8		

jmp	7	0
jle	7	1
j1	7	2
je	7	3
jne	7	4
jge	7	5
jg	7	6

Encoding Registers

- Each register has a 4-bit ID

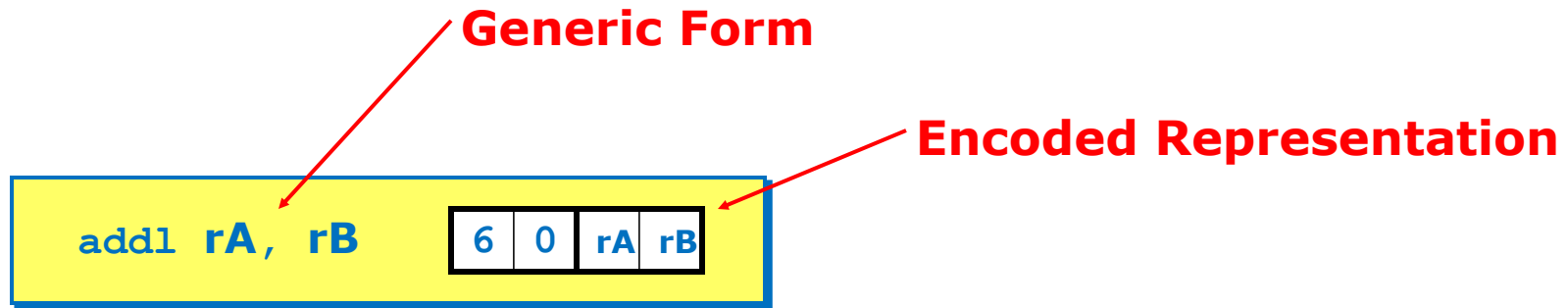
%eax	0
%ecx	1
%edx	2
%ebx	3

%esi	6
%edi	7
%esp	4
%ebp	5

- Same encoding as in IA32
-
- Register ID 15 (0xF) indicates “no register”
 - Will use this in our hardware design in multiple places

Instruction Example

■ Addition Instruction



- Add value in register rA to that in register rB
 - ▶ store result in register rB
 - ▶ note that Y86 only allows register operands
- Set condition codes based on result
- e.g., `addl %eax,%esi` Encoding: 60 06
- Two-byte encoding
 - ▶ First indicates instruction type
 - ▶ Second gives source and destination registers

Arithmetic and Logical Operations

Instruction Code **Function Code**

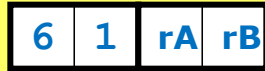
Add

`addl rA, rB`



Subtract (rA from rB)

`subl rA, rB`



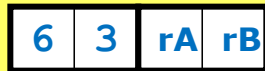
And

`andl rA, rB`



Exclusive-Or

`xorl rA, rB`



- Refer to generically as “OP1”
- Encodings differ only by “function code”
- Set condition codes as side effect

Move Operations

`rrmovl rA, rB`



Register → Register

`irmovl V, rB`



Immediate → Register

`rmmovl rA, D(rB)`



Register → Memory

`mrmovl D(rB), rA`



Memory → Register

- Like the IA32 `movl` instruction
- Simpler format for memory addresses
- Give different names to keep them distinct

Move Instruction Examples

IA32	Y86	Encoding
<code>movl \$0xabcd, %edx</code>	<code>irmovl \$0xabcd, %edx</code>	30 82 cd ab 00 00
<code>movl %esp, %ebx</code>	<code>rrmovl %esp, %ebx</code>	20 43
<code>movl -12(%ebp), %ecx</code>	<code>mrmovl -12(%ebp), %ecx</code>	50 15 f4 ff ff ff
<code>movl %esi, 0x41c(%esp)</code>	<code>rmmovl %esi, 0x41c(%esp)</code>	40 64 1c 04 00 00

<code>movl \$0xabcd, (%eax)</code>	—
<code>movl %eax, 12(%eax, %edx)</code>	—
<code>movl (%ebp, %eax, 4), %ecx</code>	—

Conditional Move Instructions

Move Unconditionally

`rrmovl rA, rB`



Move When Less or Equal

`cmovle rA, rB`



Move When Less

`cmovl rA, rB`



Move When Equal

`cmove rA, rB`



Move When Not Equal

`cmovne rA, rB`



Move When Greater or Equal

`cmovge rA, rB`



Move When Greater

`cmovg rA, rB`



- Refer to generically as “`cmovXX`”
- Encodings differ only by “function code”
- Based on values of condition codes
- Variants of `rrmovl` 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

`j1 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

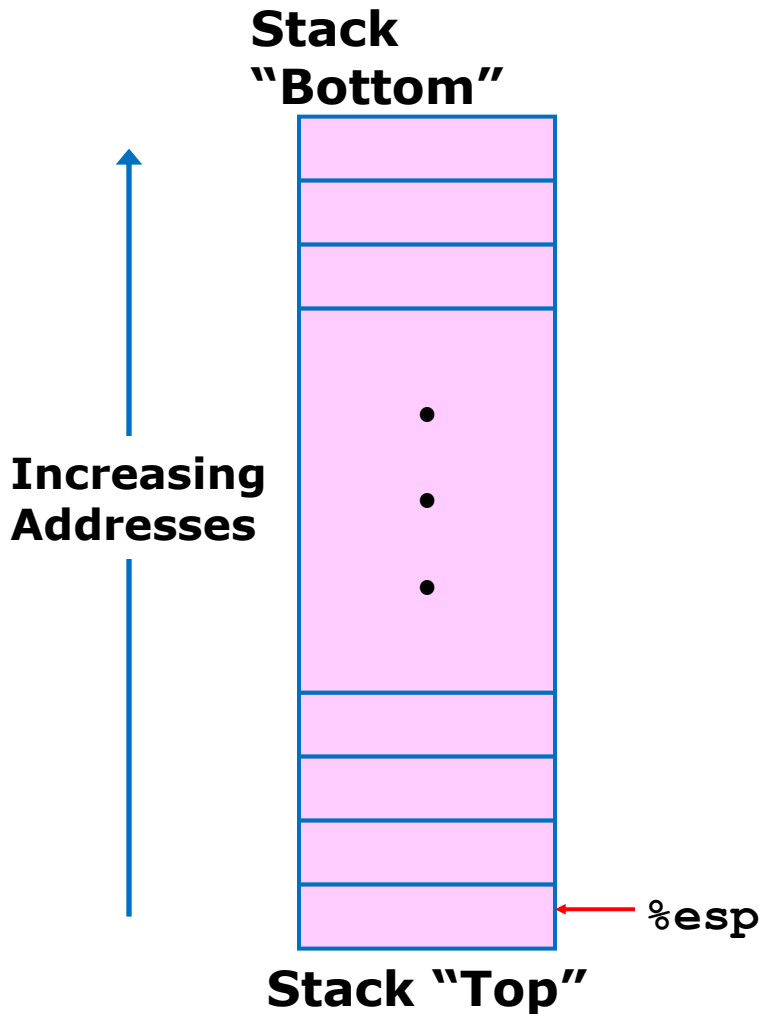
`jg Dest`

7	6
---	---

`Dest`

- Refer to generically as “jXX”
- Encodings differ only by “function code”
- Based on values of condition codes
- Same as IA32 counterparts
- Encode full destination address
 - Unlike PC-relative addressing seen in IA32

Y86 Program Stack



- Region of memory holding program data
- Used in Y86 (and IA32) for supporting procedure calls
- Stack top indicated by `%esp`
 - Address of top stack element
- Stack grows toward lower addresses
 - Top element is at highest address in the stack
 - When pushing, must first decrement stack pointer
 - After popping, increment stack pointer

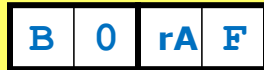
Stack Operations

pushl rA



- Decrement **%esp** by 4
- Store word from rA to memory at **%esp**
- Like IA32

popl rA



- Read word from memory at **%esp**
- Save in rA
- Increment **%esp** by 4
- Like IA32

Subroutine Call and Return

`call Dest`

8	0	Dest
---	---	------

- Push address of next instruction onto stack
- Start executing instructions at Dest
- Like IA32

`ret`

9	0
---	---

- Pop value from stack
- Use as address for next instruction
- Like IA32

Miscellaneous Instructions

`nop`



- Don't do anything

`halt`



- Stop executing instructions
- IA32 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

Mnemonic	Code
AOK	1

→ Normal operation

Mnemonic	Code
HLT	2

→ Halt instruction encountered

Mnemonic	Code
ADR	3

→ Bad address (either instruction or data)

Mnemonic	Code
INS	4

→ Invalid instruction encountered

- Desired Behavior
 - If AOK, keep going
 - Otherwise, stop program execution

Writing Y86 Code

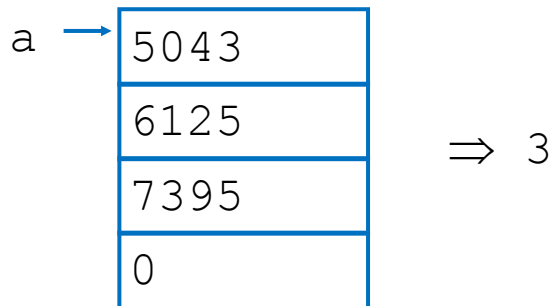
■ Try to Use C Compiler as Much as Possible

- Write code in C
- Compile for IA32 with `gcc -S -O (or -O1)`
- Transliterate into Y86

■ Coding Example

- Find number of elements in null-terminated list

```
int len1(int a[]);
```



Y86 Code Generation Example

■ First Try

- Write typical array code
- Compile with `gcc -S -O1`

```
/* Find number of elements in
   null-terminated list */
int len1(int a[])
{
    int len;
    for (len = 0; a[len]; len++)
        ;
    return len;
}
```

L5:

```
incl    %eax
cmpl    $0, (%edx,%eax,4)
jne     L5
```

■ Problem

- Hard to do array indexing on Y86
 - ▶ Since it doesn't support scaled addressing modes

Y86 Code Generation Example #2

■ Second Try

- Write with pointer code
- Compile with gcc -S -O1

```
/* Find number of elements in
   null-terminated list */
int len2(int a[])
{
    int len = 0;
    while (*a++)
        len++;
    return len;
}
```

```
.L11:
    incl    %ecx
    movl    (%edx), %eax
    addl    $4, %edx
    testl   %eax, %eax
    jne .L11
```

■ Result

- No need to do indexed addressing

Y86 Code Generation Example #3

■ IA32 Code

● Setup

```
len2:
    pushl %ebp
    movl %esp, %ebp

    movl 8(%ebp), %edx
    movl $0, %ecx
    movl (%edx), %eax
    addl $4, %edx
    testl %eax, %eax
    je .L13
```

- Need constants 1 & 4
- Store in callee-save registers

■ Y86 Code

● Setup

```
len2:
    pushl %ebp                # Save %ebp
    rrmovl %esp, %ebp        # New FP
    pushl %esi                # Save
    irmovl $4, %esi          # Constant 4
    pushl %edi                # Save
    irmovl $1, %edi          # Constant 1
    mrmovl 8(%ebp), %edx      # Get a
    irmovl $0, %ecx          # len = 0
    mrmovl (%edx), %eax       # Get *a
    addl %esi, %edx           # a++
    andl %eax, %eax           # Test *a
    je Done                  # If zero, goto Done
```

- Use `andl` to test register

Y86 Code Generation Example #4

■ IA32 Code

● Loop

```
.L11:
    incl %ecx
    movl (%edx), %eax
    addl $4, %edx
    testl %eax, %eax
    jne .L11
```

■ Y86 Code

● Loop

```
Loop:
    addl %edi, %ecx          # len++
    mrmovl (%edx), %eax     # Get *a
    addl %esi, %edx         # a++
    andl %eax, %eax         # Test *a
    jne Loop                # If !0, goto Loop
```

Y86 Code Generation Example #5

■ IA32 Code

● Finish

```
.L13:
    movl %ecx, %eax

    leave

    ret
```

■ Y86 Code

● Finish

```
Done:
    rrmovl %ecx, %eax    # return len
    popl %edi           # Restore %edi
    popl %esi           # Restore %esi
    rrmovl %ebp, %esp    # Restore SP
    popl %ebp           # Restore FP
    ret
```

Y86 Sample Program Structure #1

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

    .align 4          # Program data
array:
    . . .

Main:                # Main function
    . . .
    call len2
    . . .

len2:                # Length function
    . . .

    .pos 0x100        # Placement of stack
Stack:
```

- Program starts at address 0
- Must set up stack
 - ▶ Where located
 - ▶ Pointer values
 - ▶ Make sure we don't overwrite code!
- Must initialize data

Y86 Program Structure #2

```
init:
    irmovl Stack, %esp    # Set up SP
    irmovl Stack, %ebp    # Set up FP
    call Main             # Execute main
    halt                  # Terminate

# Array of 4 elements + terminating 0
    .align 4
array:
    .long 0x000d
    .long 0x00c0
    .long 0x0b00
    .long 0xa000
    .long 0
```

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

Y86 Program Structure #3

Main:

```
pushl %ebp
rrmovl %esp,%ebp
irmovl array,%edx
pushl %edx           # Push array
call len2            # Call len2(array)
rrmovl %ebp,%esp
popl %ebp
ret
```

- Set up call to len2
 - Follow IA32 procedure conventions
 - Push array address as argument

Assembling a Y86 Program

```
devel $ yas len.yas
```

- Generates “object code” file `len.yo`
 - ▶ Looks like disassembler output

```
0x000:                | .pos 0
0x000: 30f400010000 | init:  irmovl Stack, %esp  # Set up stack pointer
0x006: 30f500010000 |      irmovl Stack, %ebp   # Set up base pointer
0x00c: 802800000000 |      call Main           # Execute main program
0x011: 00           |      halt                # Terminate program
                        |
                        | # Array of 4 elements + terminating 0
0x014:                |      .align 4
0x014:                | array:
0x014: 0d000000      |      .long 0x000d
0x018: c0000000      |      .long 0x00c0
0x01c: 000b0000      |      .long 0x0b00
0x020: 00a00000      |      .long 0xa000
0x024: 00000000      |      .long 0
```

Simulating a Y86 Program

```
devel $ yis len.yo
```

- Instruction set simulator
 - ▶ Computes effect of each instruction on processor state
 - ▶ Prints changes in state from original

Stopped in 50 steps at PC = 0x11. Status 'HLT', CC Z=1 S=0 O=0

Changes to registers:

%eax:	0x00000000	0x00000004
%ecx:	0x00000000	0x00000004
%edx:	0x00000000	0x00000028
%esp:	0x00000000	0x00000100
%ebp:	0x00000000	0x00000100

Changes to memory:

0x00ec:	0x00000000	0x000000f8
0x00f0:	0x00000000	0x00000039
0x00f4:	0x00000000	0x00000014
0x00f8:	0x00000000	0x00000100
0x00fc:	0x00000000	0x00000011

Summary

■ Y86 Instruction Set Architecture

- Similar state and instructions as IA32
- 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
- Intel has evolved from IA32 to x86-64
 - ▶ Uses 64-bit words (including addresses)
 - ▶ Adopted some features found in RISC
 - More registers (16)
 - Less reliance on stack

CISC Instruction Sets

- Complex Instruction Set Computer
- Dominant style through mid-80's
- Stack-oriented instruction set
 - Use stack to pass arguments, save program counter
 - Explicit push and pop instructions
- Arithmetic instructions can access memory
 - `addl %eax, 12(%ebx, %ecx, 4)`
 - ▶ requires memory read and write
 - ▶ Complex address calculation
- Condition codes
 - Set as side effect of arithmetic and logical instructions
- Philosophy
 - Add instructions to perform “typical” programming tasks

RISC Instruction Sets

- Reduced Instruction Set Computer
- Internal project at IBM, later popularized by Hennessy (Stanford) and Patterson (Berkeley)
- Fewer, simpler instructions
 - Might take more to get given task done
 - Can execute them with small and fast hardware
- 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 `mrmovl` and `rmmovl`
- No Condition codes
 - Test instructions return 0/1 in register

MIPS Registers

\$0	\$0	Constant 0	\$16	\$s0	
\$1	\$at	Reserved Temp.	\$17	\$s1	
\$2	\$v0	Return Values	\$18	\$s2	
\$3	\$v1		\$19	\$s3	Callee Save Temporaries: May not be overwritten by called procedures
\$4	\$a0	Procedure arguments	\$20	\$s4	
\$5	\$a1		\$21	\$s5	
\$6	\$a2		\$22	\$s6	
\$7	\$a3		\$23	\$s7	
\$8	\$t0	Caller Save Temporaries: May be overwritten by called procedures	\$24	\$t8	Caller Save Temp
\$9	\$t1		\$25	\$t9	
\$10	\$t2		\$26	\$k0	
\$11	\$t3		\$27	\$k1	Reserved for Operating Sys
\$12	\$t4		\$28	\$gp	Global Pointer
\$13	\$t5		\$29	\$sp	Stack Pointer
\$14	\$t6		\$30	\$s8	Callee Save Temp
\$15	\$t7		\$31	\$ra	Return Address

CISC Instruction Sets

- Complex Instruction Set Computer
 - Dominant style through mid-80's
- Stack-oriented instruction set
 - Use stack to pass arguments, save program counter
 - Explicit push and pop instructions
- Arithmetic instructions can access memory
 - `addl %eax, 12(%ebx, %ecx, 4)`
 - ▶ requires memory read and write
 - ▶ Complex address calculation
- Condition codes
 - Set as side effect of arithmetic and logical instructions
- Philosophy
 - Add instructions to perform “typical” programming tasks

RISC Instruction Sets

- Reduced Instruction Set Computer
 - Internal project at IBM, later popularized by Hennessy (Stanford) and Patterson (Berkeley)
- Fewer, simpler instructions
 - Might take more to get given task done
 - Can execute them with small and fast hardware
- 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 `mrmovl` and `rmmovl`
- No Condition codes
 - Test instructions return 0/1 in register

Example: MIPS – Registers

\$0	\$0	Constant 0	\$16	\$s0	
\$1	\$at	Reserved Temp.	\$17	\$s1	
\$2	\$v0	Return Values	\$18	\$s2	Callee Save Temporaries: May not be overwritten by called procedures
\$3	\$v1		\$19	\$s3	
\$4	\$a0	Procedure arguments	\$20	\$s4	
\$5	\$a1		\$21	\$s5	
\$6	\$a2		\$22	\$s6	
\$7	\$a3		\$23	\$s7	Caller Save Temp
\$8	\$t0	Caller Save Temporaries: May be overwritten by called procedures	\$24	\$t8	
\$9	\$t1		\$25	\$t9	
\$10	\$t2		\$26	\$k0	Reserved for Operating Sys
\$11	\$t3		\$27	\$k1	
\$12	\$t4		\$28	\$gp	Global Pointer
\$13	\$t5		\$29	\$sp	Stack Pointer
\$14	\$t6		\$30	\$s8	Callee Save Temp
\$15	\$t7		\$31	\$ra	Return Address

Example: MIPS – Instructions

R-R

Op	Ra	Rb	Rd	00000	Fn
----	----	----	----	-------	----

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

R-I

Op	Ra	Rb	Immediate
----	----	----	-----------

`addu $3,$2, 3145` # Immediate add: $\$3 = \$2 + 3145$

`sll $3,$2,2` # Shift left: $\$3 = \$2 \ll 2$

Branch

Op	Ra	Rb	Offset
----	----	----	--------

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

Load/Stor

^e Op	Ra	Rb	Offset
-----------------	----	----	--------

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

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

CISC vs. RISC

■ Original Debate

- Strong opinions!
- CISC proponents: easy (?) for compilers, fewer code bytes
- RISC proponents: easy (?) for 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
 - ▶ Installed base / code compatibility more important
- For embedded processors, RISC makes sense
 - ▶ Smaller, cheaper, less power
 - ▶ Most cell phones use ARM processor