Organisation und Architektur von Rechnern

Lecture 09

Instructor:

Reinhard v. Hanxleden

http://www.informatik.uni-kiel.de/rtsys/teaching/v-sysinf2

These slides are used with kind permission from the Carnegie Mellon University

The 5 Minute Review Session

1. How does procedure calling differ in IA32 and x86-64?
2. How are one-dimensional arrays stored in memory?
3. How about multi-dimensional arrays?
4. What are `structs` in C?
5. How are `structs` mapped to memory?

Computer Architecture – Outline

- **Background**
  - Instruction sets
  - Logic design
- **Sequential Implementation**
  - A simple, but not very fast processor design
- **Pipelining**
  - Get more things running simultaneously
- **Pipelined Implementation**
  - Make it work
- **Advanced Topics**
  - Performance analysis
  - High performance processor design

Computer Architecture – Coverage

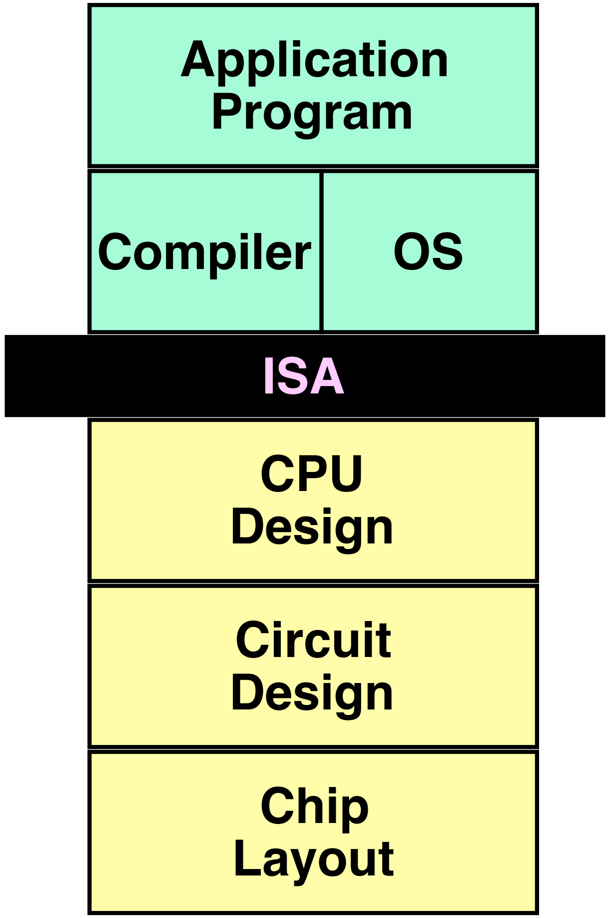
- **Our Approach**
  - Work through designs for particular instruction set
    - Y86---a simplified version of the Intel IA32 (a.k.a. x86).
    - If you know one, you more-or-less know them all
  - Work at "microarchitectural" level
    - Assemble basic hardware blocks into overall processor structure
      - Memories, functional units, etc.
    - Surround by control logic to make sure each instruction flows through properly
  - Use simple hardware description language to describe control logic
    - Can extend and modify
    - Test via simulation

Computer Architecture – Schedule

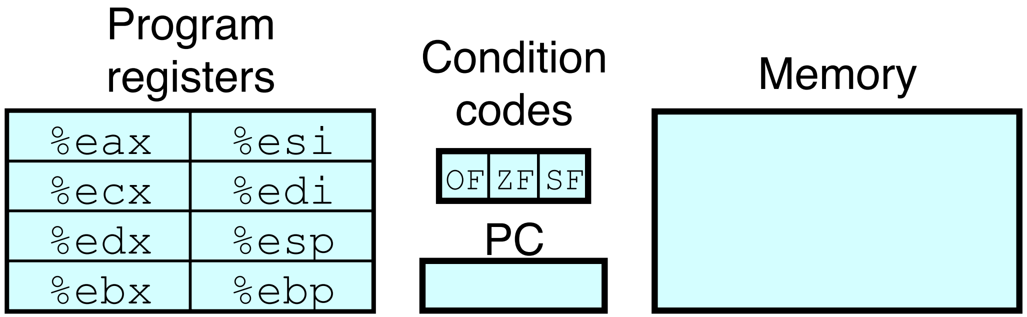
- **Week #1**
  - Instruction set architecture
  - Logic design
- **Week #2**
  - Sequential implementation
  - Pipelining and initial pipelined implementation
- **Week #3**
  - Making the pipeline work
  - Modern processor design

Instruction Set Architecture

- **Assembly Language View**
  - Processor state
    - Registers, memory, …
  - Instructions
    - `addl`, `movl`, `leal`, …
    - 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
    - OF: Overflow   ZF: Zero   SF:Negative
- Program Counter
  - Indicates address of instruction
- Memory
  - Byte-addressable storage array
  - Words stored in little-endian byte order

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

Encoding Registers

- **Each register has 4-bit ID**

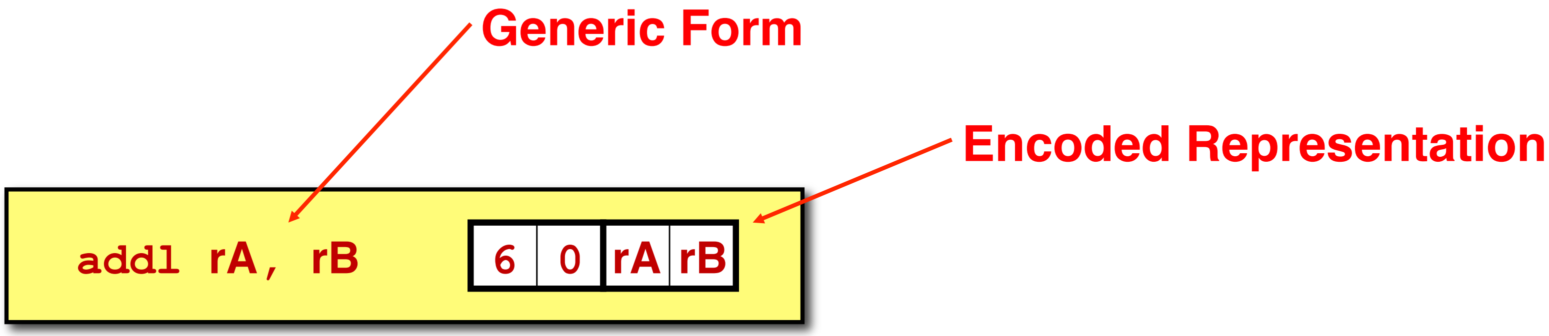
`%eax`	0
`%ecx`	1
`%edx`	2
`%ebx`	3

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

  - Same encoding as in IA32
- **Register ID 8 indicates “no register”**
  - Will use this in our hardware design in multiple places

Instruction Example

- **Addition Instruction**



Generic Form

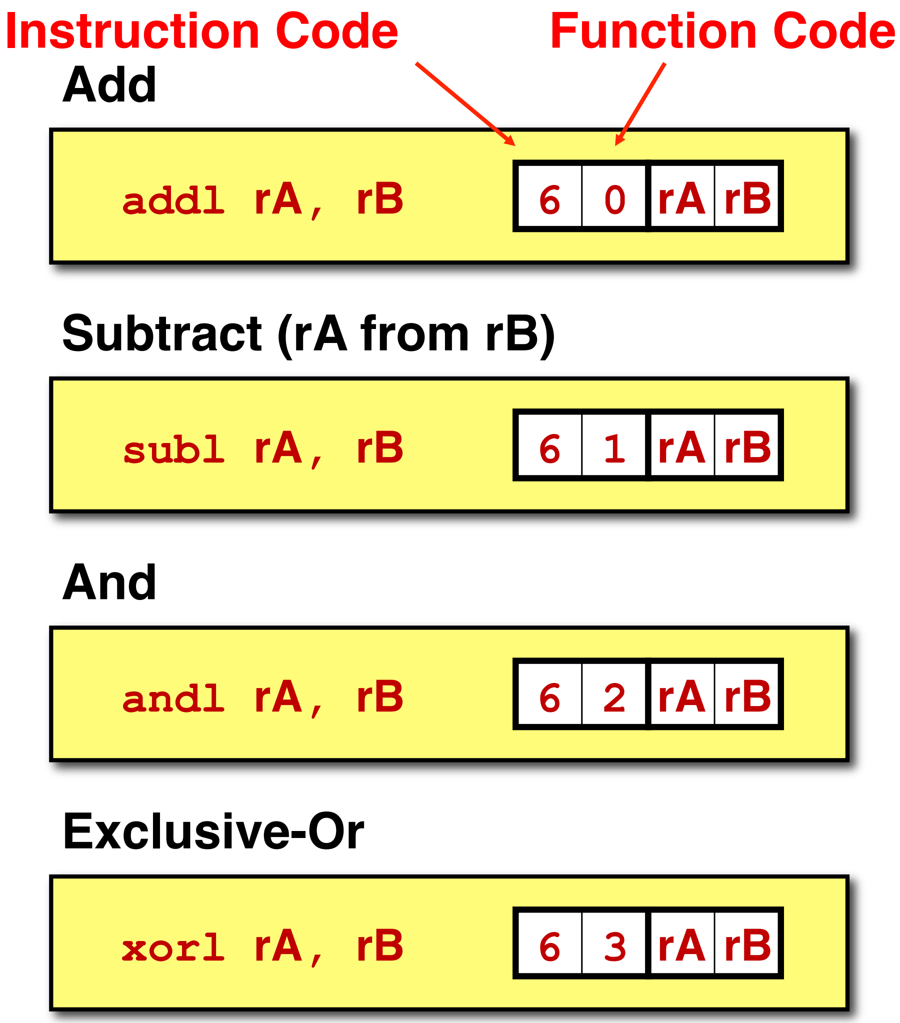
Encoded Representation

`addl rA, rB`

6	0	rA	rB

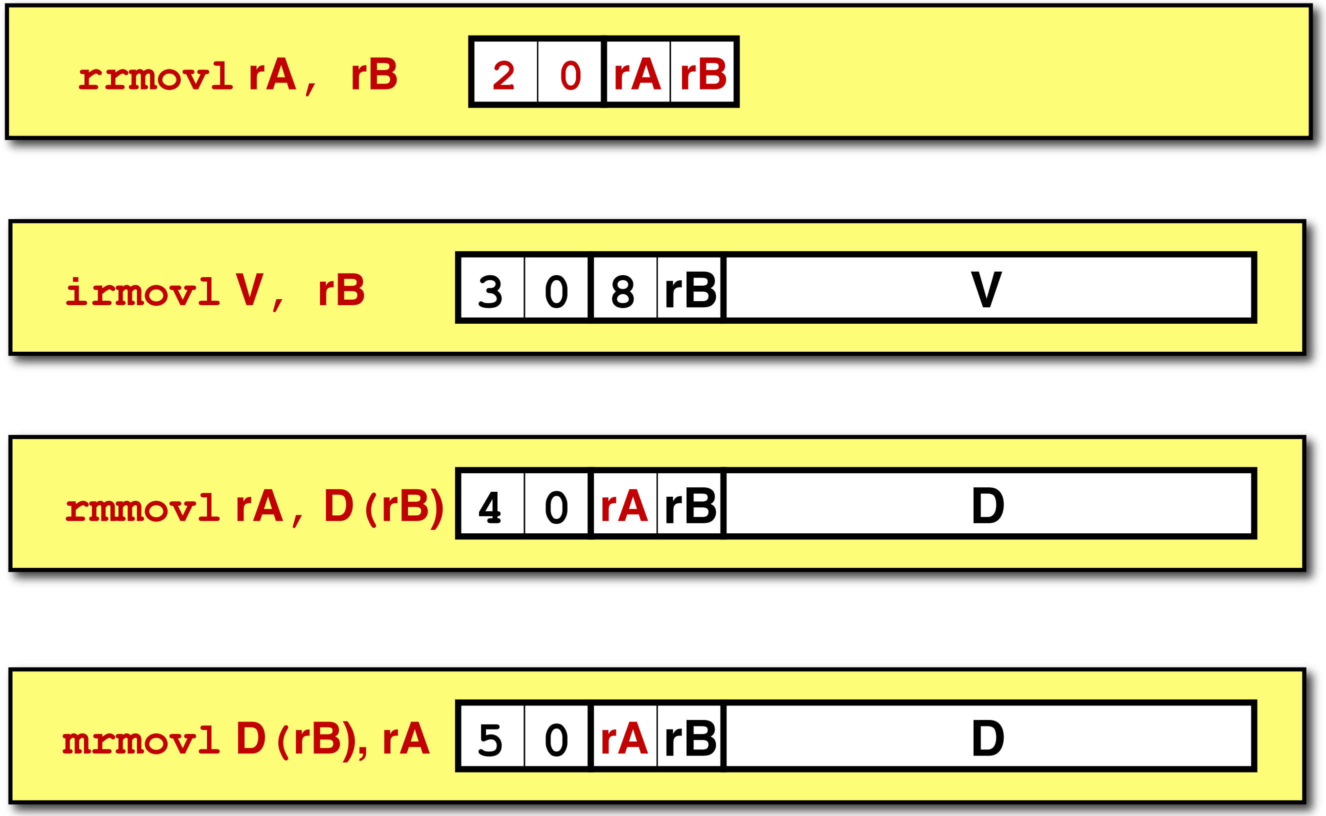
- Add value in register rA to that in register rB
  - Store result in register rB
  - Note that Y86 only allows addition to be applied to register data
- Set condition codes based on result
- e.g., `addl %eax,%esi` Encoding: `60 06`
- Two-byte encoding
  - First indicates instruction type
  - Second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

Arithmetic and Logical Operations



- Refer to generically as "`OPl`"
- Encodings differ only by "function code"
  - Low-order 4 bits in first instruction word
- Set condition codes as side effect

Move Operations



`rrmovl rA, rB`	2 | 0 | rA | rB	**Register --> Register**
`irmovl V, rB`	3 | 0 | 8 | rB | V	**Immediate --> Register**
`rmmovl rA, D(rB)`	4 | 0 | rA | rB | D	**Register --> Memory**
`mrmovl D(rB), rA`	5 | 0 | rA | rB | D	**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
`movl $0xabcd, %edx`	`irmovl $0xabcd, %edx`	`30 82 cd ab 00 00`
`movl %esp, %ebx`	`rrmovl %esp, %ebx`	`20 43`
`movl -12(%ebp),%ecx`	`mrmovl -12(%ebp),%ecx`	`50 15 f4 ff ff ff`
`movl %esi,0x41c(%esp)`	`rmmovl %esi,0x41c(%esp)`	`40 64 1c 04 00 00`

`movl $0xabcd, (%eax)`	—
`movl %eax, 12(%eax,%edx)`	—
`movl (%ebp,%eax,4),%ecx`	—

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

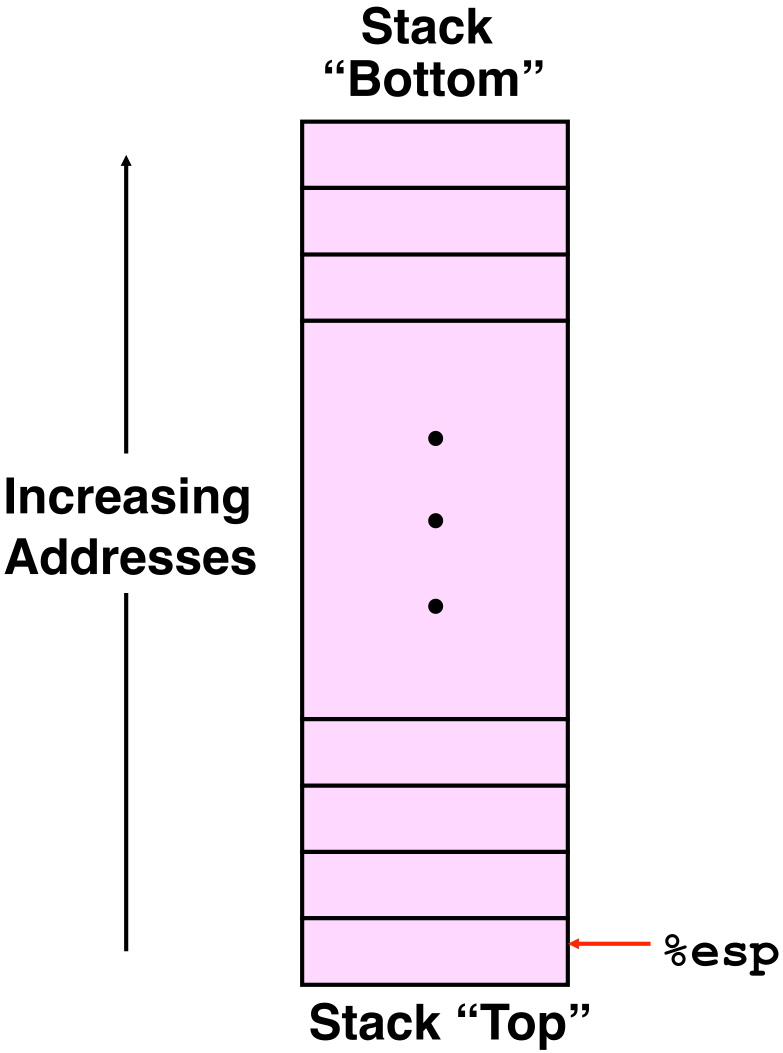
`jge` **Dest**	7	5	Dest

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 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 lowest address in the stack
  - When pushing, must first decrement stack pointer
  - When popping, increment stack pointer

Stack Operations

`pushl rA`

a	0	rA	8

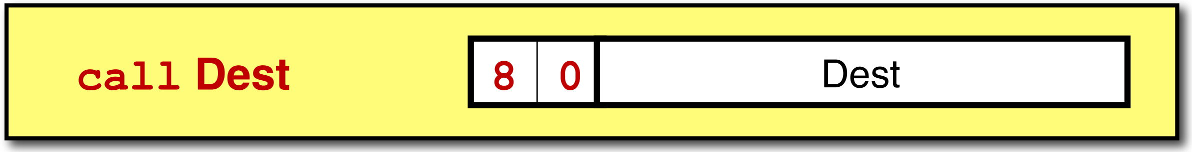
- Decrement `%esp` by 4
- Store word from rA to memory at `%esp`
- Like IA32

`popl rA`

b	0	rA	8

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

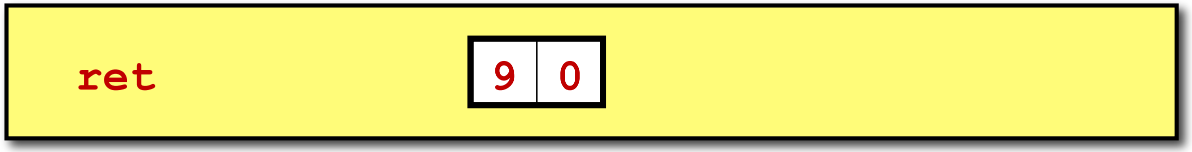
Subroutine Call and Return



`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`	0	0

- Don’t do anything

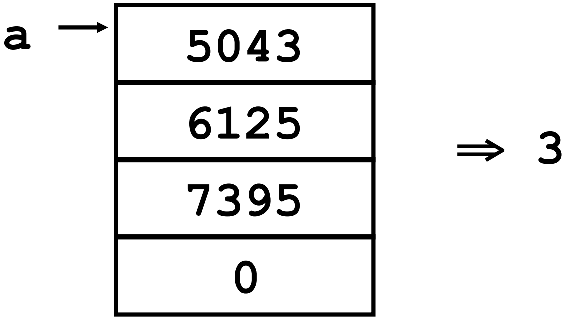
`halt`	1	0

- Stop executing instructions
- IA32 has comparable instruction, but can’t execute it in user mode
- We will use it to stop the simulator

Writing Y86 Code

- **Try to Use C Compiler as Much as Possible**
  - Write code in C
  - Compile for IA32 with `gcc -S`
  - 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

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

  - Compile with `gcc -O2 -S`

- **Problem**
  - Hard to do array indexing on Y86
    - Since don't have scaled addressing modes

```
L18:
   incl %eax
   cmpl $0,(%edx,%eax,4)
   jne L18
```

Y86 Code Generation Example #2

- **Second Try**
  - Write with pointer code

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

- Compile with `gcc -O2 -S`

- **Result**
  - Don’t need to do indexed addressing

```
L24:
   movl (%edx),%eax
   incl %ecx
L26:
   addl $4,%edx
   testl %eax,%eax
   jne L24
```

Y86 Code Generation Example #3

- **IA32 Code**
  - Setup

```
len2:
   pushl %ebp
   xorl %ecx,%ecx
   movl %esp,%ebp
   movl 8(%ebp),%edx
   movl (%edx),%eax
   jmp L26
```

- **Y86 Code**
  - Setup

```
len2:
   pushl %ebp             # Save %ebp
   xorl %ecx,%ecx         # len = 0
   rrmovl %esp,%ebp       # Set frame
   mrmovl 8(%ebp),%edx# Get a
   mrmovl (%edx),%eax # Get *a
   jmp L26                # Goto entry
```

Y86 Code Generation Example #4

- **IA32 Code**
  - Loop + Finish

```
L24:
   movl (%edx),%eax
   incl %ecx

L26:
   addl $4,%edx

   testl %eax,%eax
   jne L24
   movl %ebp,%esp
   movl %ecx,%eax
   popl %ebp
   ret
```

- **Y86 Code**
  - Loop + Finish

```
L24:
   mrmovl (%edx),%eax  # Get *a
   irmovl $1,%esi
   addl %esi,%ecx      # len++
L26:                   # Entry:
   irmovl $4,%esi
   addl %esi,%edx      # a++
   andl %eax,%eax      # *a == 0?
   jne L24             # No--Loop
   rrmovl %ebp,%esp    # Pop
   rrmovl %ecx,%eax    # Rtn len
   popl %ebp
   ret
```

Y86 Program Structure

```
   irmovl Stack,%esp     # Set up stack
   rrmovl %esp,%ebp      # Set up frame
   irmovl List,%edx
   pushl %edx            # Push argument
   call len2             # Call Function
   halt                  # Halt
.align 4
List:                    # List of elements
   .long 5043
   .long 6125
   .long 7395
   .long 0

# Function
len2:
   . . .

# Allocate space for stack
.pos 0x100
Stack:
```

- Program starts at address 0
- Must set up stack
  - Make sure don’t overwrite code!
- Must initialize data
- Can use symbolic names

Assembling Y86 Program

```
unix> yas eg.ys
```

- Generates “object code” file `eg.yo`
  - Actually looks like disassembler output

```
0x000: 308400010000 | irmovl Stack,%esp       # Set up stack
0x006: 2045         | rrmovl %esp,%ebp        # Set up frame
0x008: 308218000000 | irmovl List,%edx
0x00e: a028         | pushl %edx              # Push argument
0x010: 8028000000   | call len2               # Call Function
0x015: 10           | halt                    # Halt
0x018:              | .align 4
0x018:              | List:                   # List of elements
0x018: b3130000     | .long 5043
0x01c: ed170000     | .long 6125
0x020: e31c0000     | .long 7395
0x024: 00000000     | .long 0
```

Simulating Y86 Program

```
unix> yis eg.yo
```

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

```
Stopped in 41 steps at PC = 0x16.  Exception 'HLT', CC Z=1 S=0 O=0
Changes to registers:
%eax:                   0x00000000     0x00000003
%ecx:                   0x00000000     0x00000003
%edx:                   0x00000000     0x00000028
%esp:                   0x00000000     0x000000fc
%ebp:                   0x00000000     0x00000100
%esi:                   0x00000000     0x00000004

Changes to memory:
0x00f4:                 0x00000000     0x00000100
0x00f8:                 0x00000000     0x00000015
0x00fc:                 0x00000000     0x00000018
```

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 is moving away from IA32
    - Does not allow enough parallel execution
    - Introduced IA64
      - 64-bit word sizes (overcome address space limitations)
      - Radically different style of instruction set with explicit parallelism
      - Requires sophisticated compilers

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
$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	
$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
```

R-I

Op	Ra	Rb	Immediate

```
addu $3,$2, 3145    # Immediate add: $3 = $2+3145
sll $3,$2,2          # Shift left: $3 = $2 << 2
```

Branch

Op	Ra	Rb	Offset

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

Load/Store

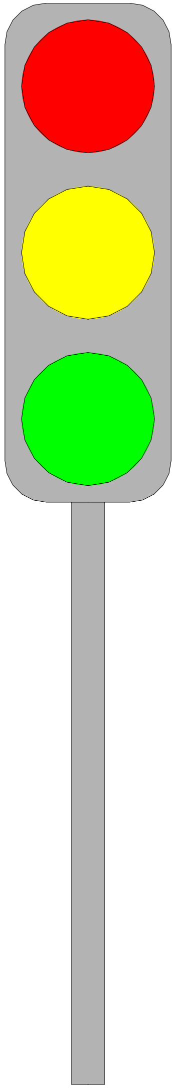
Op	Ra	Rb	Offset

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

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
  - For embedded processors, RISC makes sense
    - Smaller, cheaper, less power

Rückmeldung zu dieser Vorlesung …



a) Gut verstanden – Stoff leicht

b) Einigermaßen verstanden – Stoff ok

c) Wenig verstanden – Stoff schwer