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
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
Which is IA32?
- CISC

Which is Y86?
- Includes attributes of both.
  - CISC
    - Condition codes
    - Variable length instructions
    - Stack intensive procedure linkages
  - RISC
    - Load-store architecture
    - Regular encoding

Which is better: RISC or CISC?
Y86 Exceptions

What happens when an invalid assembly instruction is found?

- How would this happen?
- This generates an exception.

In Y86 an exception halts the machine, it stops executing.

- On a real system, this would be handled by the OS and only the current process would be terminated.

What are some possible causes of exceptions?

- Invalid operation
- Divide by 0
- sqrt of negative number
- Memory access error (address too large)
- Hardware error

Y86 handles 3 types of exceptions: HLT instruction executed

- Invalid address encountered
- Invalid instruction encountered
- In each case the status is set
The CPU uses machine language to perform all its operations.

Assembly is a much more readable translation of machine language, and it is what we work with if we need to see what the computer is doing.

Many different kinds of assembly languages.

Machine code (pure numbers) is generated by translating each instruction into binary numbers that the CPU uses.

This process is called “assembling.”

Conversely, we can take assembled code and disassemble it into (mostly) human readable assembly language.
Run the "subscribe" command on an stdlinux machine and choose the Y86SIM option (#18). Remember that you need to log out and in again after doing that. Once that's done, the following directories are added to your $PATH:

- /usr/local/sim/misc
- /usr/local/sim/pipe
- /usr/local/sim/seq

The example code was assembled during the build process and is in /usr/local/sim/y86-code.

**HOW TO:**

- %yas prog.ys
  - Assembles the program
  - Creates a *.yo file
- %yis prog.yo
  - Simulator – gives output and changes
- %ssim –g prog.yo &

**SimGuide**

link→ http://csapp.cs.cmu.edu/public/simguide.pdf
Run Y86 program

```
irmovl $55,%edx
rrmovl %edx, %ebx
irmovl Array, %eax
rmmovl %ebx,4(%eax)
mrmovl 0(%eax),%ecx
halt

.align 4
Array:
.long 0x6f
.long 0x84
```

```
% yas y86prog1.ys
% yis y86prog1.yo
Stopped in 6 steps at PC = 0x1a.
Status 'HLT'
CC Z=1 S=0 O=0
Changes to registers:
%eax: 0x00000000 0x0000001c
%ecx: 0x00000000 0x0000006f
%edx: 0x00000000 0x00000037
%ebx: 0x00000000 0x00000037

Changes to memory:
0x0020: 0x00000084 0x00000037
```
# Y86 Assembler directives

<table>
<thead>
<tr>
<th>Directive</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>.pos number</code></td>
<td>Subsequent lines of code start at address <strong>number</strong></td>
</tr>
<tr>
<td><code>.align number</code></td>
<td>Align the next line to a <strong>number</strong>-byte boundary</td>
</tr>
<tr>
<td><code>.long number</code></td>
<td>Put <strong>number</strong> at the current address in memory</td>
</tr>
</tbody>
</table>

- These can be used to set up memory in various places in the address space
- `.pos` can put sections of code in different places in memory
- `.align` should be used before setting up a static variable
- `.long` can be used to initialize a static variable
Simulator program code

<table>
<thead>
<tr>
<th>File</th>
<th>y86progl.yo</th>
</tr>
</thead>
</table>
| 0x0      | 30f237000000 | irmovl $55,%edx  
| 0x6      | 2023        | rrmovl %edx, %ebx  
| 0x8      | 30f01c000000 | irmovl Array, %eax  
| 0xe      | 403004000000 | rrmovl %ebx,4(%eax)  
| 0x14     | 501000000000 | mrmovl 0(%eax),%ecx  
| 0x1a     | 00          | halt  
| 0x1c     | 6f000000    | .long 0x6f  
| 0x20     | 84000000    | .long 0x84  |
C program to IA32 and Y86

Computes the sum of an integer array

```c
int Sum(int *Start, int Count)
{
    int sum = 0;
    while (Count)
    {
        sum += *Start;
        Start++;
        Count--;
    }
}
```

ASSEMBLY COMPARISON ON NEXT SLIDE

Why not using array indexing?
No, scaled addressing modes in Y86

Uses stack and frame pointers

For simplicity, does not follow IA32 convention of having some registers designated as callee-save registers (convention so adopt or ignore as we please)
IA32/Y86 comparison

```assembly
IA32 code

```
int Sum(int *Start, int Count)
{
    Sum:
    pushl %ebp
    movl %esp,%ebp
    movl 8(%ebp),%ecx
    ecx = start
    movl 12(%ebp),%edx
    edx = count
    xorl %eax,%eax
    sum = 0
    testl %edx,%edx
    je .L34
    .L35:
    addl (%ecx),%eax
    add *Start to sum
    addl $4,%ecx
    Start++
    decl %edx
    Count--
    jnz .L35
    Stop when 0
    .L34:
    movl %ebp,%esp
    popl %ebp
    ret
```

```assembly
Y86 code

```
int Sum(int *Start, int Count)
{
    Sum:
    pushl %ebp
    rrmovl %esp,%ebp
    mrmovl 8(%ebp),%ecx
    ecx = start
    mrmovl 12(%ebp),%edx
    edx = count
    xorl %eax,%eax
    sum = 0
    andl %edx,%edx
    Set condition codes
    je End
    Loop:
    mrmovl (%ecx),%esi
    get *Start
    addl %esi,%eax
    add to sum
    irmovl $4,%ebx
    Start++
    addl %ebx,%ecx
    Count--
    irmovl $-1,%ebx
    Stop when 0
    jne Loop
    End:
    rrmovl %ebp,%esp
    popl %ebp
    ret
```

Figure 4.6: Comparison of Y86 and IA32 assembly programs. The `Sum` function computes the sum of an integer array. The Y86 code differs from the IA32 mainly in that it may require multiple instructions to perform what can be done with a single IA32 instruction.
The Y86 can also read and write to memory, which is just a huge array of bytes. However, one needs to be careful with the Y86 simulator concerning memory as Y86 programs reference memory using virtual addresses. A programmer does not want to overwrite the code of a program, as the data and code share the same memory space. Therefore, the stack should be set far enough away from the code, or devastating results could happen to the program.
Memory is one contiguous chunk that starts at address 0x0. All programs start executing at address 0x0. Initialized data is interleaved with the instructions in memory as defined in the source assembly. Similarly, the initial value for the stack pointer (%esp) is explicitly set by the program.

Note, that for any test programs you write, you must start the text at address 0x0 (i.e., .pos 0x0 before the first instruction that will be executed), and if your program requires a stack you must explicitly set the stack in your source program.