Summer 2013
CSE2421 Systems 1
Introduction to Low-Level Programming and Computer Organization
Kitty Reeves
MTWR 10:20am-12:25pm
The CPU uses machine language to perform all its operations.

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.

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.

There are many different kinds of assembly languages; we'll focus on the Y86/IA32 language as defined in the text and on our system (also SPARC and MIPS).
Assembly Operations

- Perform arithmetic function on register or memory data
- Transfer data between memory and register
  - Load data from memory into register (read)
  - Store register data into memory (write)
- Transfer control
  - Unconditional jumps to/from procedures (calls)
  - Conditional branches (if, switch, for, while, etc)
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
Machine-level code

ISA – instruction set architecture

Format and behavior of a machine level program

Defines:
- The processor state (see the CPU fetch-execute cycle)
- The format of the instructions
- The effect of each of these instructions on the state

Abstractions
- Instruction executed “in sequence”
  - Technically defined to be completing one instruction before starting the next
  - Pipelining
  - Concurrent execution (but not really)
  - Jumps and calls
- Memory addresses are virtual addresses
  - Very large byte-addressable array
  - Address space managed by the OS (virtual → physical)
  - Contains both executable code of the program AND its data
    - Run-time stack
    - Block of memory for user (global and heap)
An **instruction cycle** is the basic operation cycle of a computer. It is the process by which a computer retrieves a program instruction from its memory, determines what actions the instruction requires, and carries out those actions. This cycle is repeated continuously by the central processing unit (CPU), from bootup to when the computer is shut down.

1. Fetching the instruction
2. Decode the instruction
3. Memory and addressing issues
4. Execute the instruction
CPU Fetch-Execute cycle

- **Initiating the cycle**
  - The cycle starts immediately when power is applied to the system using an initial PC value that is predefined for the system architecture (in Intel IA-32 CPUs, for instance, the predefined PC value is 0xffffffff0). Typically this address points to instructions in a read-only memory (ROM) which begin the process of loading the operating system. (That loading process is called booting.)

- **Fetch cycle**
  - Step 1 of the Instruction Cycle is called the Fetch Cycle. These steps are the same for each instruction. The fetch cycle processes the instruction from the instruction word which contains an opcode.

- **Decode**
  - Step 2 of the instruction Cycle is called the decode. The opcode fetched from the memory is being decoded for the next steps and moved to the appropriate registers.

- **Read the effective address**
  - Step 3 is deciding which operation it is. If this is a Memory operation - in this step the computer checks if it's a direct or indirect memory operation:
    - Direct memory instruction - Nothing is being done.
    - Indirect memory instruction - The effective address is being read from the memory.
  - If this is a I/O or Register instruction - the computer checks its kind and execute the instruction.

- **Execute cycle**
  - Step 4 of the Instruction Cycle is the Execute Cycle. These steps will change with each instruction.
Machine code vs C code

- Program Counter (PC)
  - Register %eip (X86-64)
  - Address in memory of the next instruction to be executed

- Integer Register File
  - Contains eight named locations for storing 32-bit values
    - Can hold addresses (C pointers) or integer data
    - Have other special duties

- Condition Code registers
  - Hold status information
    - About arithmetic or logical instruction executed
      - CF (carry flag)
      - OF (overflow flag)
      - SF (sign flag)
      - ZF (zero flag)

- Floating point registers
Machine Instruction Example

- **C code**
  - Add two signed integers

- **Assembly**
  - Add 2 4-byte integers

- **Operands**
  - X: register %eax
  - Y: memory M[%ebp+8]
  - T: register %eax
  - Return function value in %eax

- **Object code**
  - 3 byte instruction
  - Stored at address: 0x????????

```c
int t = x + y;
addl 8(%ebp),%eax
```

03 45 08
Y86: A simpler instruction set

- IA32 has a lot more instructions
- IA32 has a lot of quirks
- Y86 is a subset of IA32 instructions
- Y86 has a simpler encoding scheme than IA32
- Y86 is easier to reason about
  - hardware
  - first time programming in assembly language
About Y86

- The Y86 has
  - 8 32-bit registers with the same names as the IA32 32-bit registers
  - 3 condition codes: ZF, SF, OF
    - no carry flag
    - interprets integers as signed
  - a program counter (PC)
  - a program status byte: AOK, HLT, ADR, INS
  - memory: up to 4 GB to hold program and data
- The Y86 does not have:
  - floating point registers or instructions

http://y86tutoring.wordpress.com/
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.
YIS, YAS and the Y86 simulator

- Run the "subscribe" command on an stdlinux machine and choose the Y86SIM option (#18). Remember that you need to log out and log back 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`
    - Instruction set simulator – gives output and changes
    - Usage: yis code_file [max_steps]
  - `%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

y86prog1.ys
Y86 Simulator program code
Y86 Simulator program code

% more y86prog1.yo

0x000: 30f237000000 | irmovl $55,%edx  # first line
0x006: 2023 | rrmovl %edx, %ebx  # this symbol denotes
0x008: 30f01c000000 | irmovl Array, %eax  # a comment
0x00e: 403004000000 | rmmovl %ebx,4(%eax)
0x014: 501000000000 | mrmovl 0(%eax),%ecx
0x01a: 00 | halt
0x01c: | .align 4
0x01c: | Array:
0x01c: 6f000000 | .long 0x6f
0x020: 84000000 | .long 0x84

# blank lines okay
# notice the address of Array
# check how mrmovl is encoded
Y86 Simulator

- Contents of memory
- Processor State
  - The fetch-execute loop
- Register file
- Status
- Condition Codes

![Y86 Simulator Interface](image)
Y86 programmer-visible state

- Y86 is an assembly language instruction set simpler than but similar to IA32; but not as compact (as we will see)
- The Y86 has:
  - 8 32-bit registers with the same names as the IA32 32-bit registers
  - 3 condition codes: ZF, SF, OF
    - no carry flag - interpret integers as signed
  - a program counter (PC)
    - Holds the address of the instruction currently being executed
  - a program status byte: AOK, HLT, ADR, INS
    - State of program execution
  - memory: up to 4 GB to hold program and data (4096 = 2^12)

<table>
<thead>
<tr>
<th>RF: Program registers</th>
<th>CC: Condition codes</th>
<th>Stat: Program Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>%esi</td>
<td>ZF</td>
</tr>
<tr>
<td>%ecx</td>
<td>%edi</td>
<td>PC</td>
</tr>
<tr>
<td>%edx</td>
<td>%esp</td>
<td></td>
</tr>
<tr>
<td>%ebx</td>
<td>%ebp</td>
<td></td>
</tr>
</tbody>
</table>

DMEM: Memory
Learning Y86

- Assembler directives
- Status conditions and Exceptions
- Instructions
  - Operations
  - Branches
  - Moves
- Addressing Modes
- Stack operations
- Subroutine call/return
- How to encode each instruction
Y86 Assembler directives

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

- Normal operation
- Halt instruction encountered
- Bad address (either instruction or data) encountered
- Invalid instruction encountered

**Desired Behavior**

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

Halt (HLT) should always be the result of the yis command.
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
Y86 Instructions

- Each accesses and modifies some part(s) of the program state
- Largely a subset of the IA32 instruction set
  - Includes only 4-byte integer operations \( \rightarrow \) “word”
  - Has fewer addressing modes
  - Smaller set of operations
- Format
  - 1–6 bytes of information read from memory
    - Can determine the type of instruction from first byte
    - Can determine instruction length from first byte
    - Not as many instruction types
    - Simpler encoding than with IA32
- Registers
  - \( rA \) or \( rB \) represent one of the registers (0-7)
  - 0xF denotes no register (when needed)
  - No partial register options (must be a byte)
Move operation

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>irmovl V,R</td>
<td>Reg[R] ← V</td>
<td>Immediate-to-register move</td>
</tr>
<tr>
<td>rrmovl rA,rB</td>
<td>Reg[rB] ← Reg[rA]</td>
<td>Register-to-register move</td>
</tr>
</tbody>
</table>

- **irmovl** is used to place known numeric values (labels or numeric literals) into registers
- **rrmovl** copies a value between registers
- **rmmovl** stores a word in memory
- **mrmovl** loads a word from memory
- **rrmovl** and **mrmovl** are the only instructions that access memory - Y86 is a load/store architecture
Move operation

- Different opcodes for 4 types of moves
  - register to register (opcode = 2)
    - Notice conditional move has opcode 2 as well
  - immediate to register (opcode = 3)
  - register to memory (opcode = 4)
  - memory to register (opcode = 5)

- The only memory addressing mode is base register + displacement
- Memory operations always move 4 bytes (no byte or word memory operations i.e. no 8/16-bit move)
- Source or destination of memory move must be a register.

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

**CORRECTION** = F
Simple Addressing Modes

- Normal = (R) = Mem[Reg[R]]
  - Register Reg specifies memory address
  - Example: `movl (%ecx),%eax`

- Displacement = D(R) = Mem[Reg[R]+D]
  - Register R specifies start of memory region
  - Constant displacement D specifies offset
  - In bytes!
  - Example: `movl 8(%ebp),%edx`
Supported Operations

- **OP1** (opcode = 6)
  - Only take registers as operands
  - Only work on 32 bits
  - Note: no “or” and “not” ops
  - Only instructions to set CC
    - Starting point ZF=1, SF=0, OF=0

- Arithmetic instructions
  - addl rA, rB R[rB] ← R[rB] + R[rA]
  - subl rA, rB R[rB] ← R[rB] − R[rA]
  - andl rA, rB R[rB] ← R[rB] & R[rA]
  - xorl rA, rB R[rB] ← R[rB] ^ R[rA]

# y86cc.ys
.pos 0x0
irmovl $1, %eax
irmovl 0, %ebx
irmovl $1, %ecx
addl %eax, %eax
andl %ebx, %ebx
subl %eax, %ecx
irmovl $0x7ffffff, %edx
addl %edx, %edx
halt

* Notice & ^ are bit operations
* Order of operands important esp subl
Y86 condition codes

- **ZF → Zero Flag**
  - If the result of the current ALU operation is zero, then Z is 1.
  - If the result of the operation is not zero, then Z is 0.

- **SF → Sign Flag**
  - Reflects the most significant bit of the result of the current ALU operation i.e. the sign bit indicates a negative result.

- **OF → Overflow Flag**
  - For arithmetic operations (addl and subl), this bit is 1 if the current operation caused a two's complement overflow (either positive or negative) and is 0 otherwise.
  - For logical operations (andl and xorl), this bit is always set to 0 i.e. there can’t be overflow with these operations.
Jump instructions (opcode = 7)
- fn = 0 for unconditional jump
- fn =1-6 for <=  <  =  !=  >=  >
- Refer to generically as “jXX”
- Encodings differ only by “function code”
- Based on values of condition codes

FYI for later
Same as IA32 counterparts
Encode full destination address
  - Unlike PC-relative addressing seen in IA32
## Jump instruction types

### Unconditional jumps
- jmp Dest  \[ \text{PC} \leftarrow \text{Dest} \]

### Conditional jumps
- jle Dest  \[ \text{PC} \leftarrow \text{Dest} \text{ iff last result} \leq 0 \]
  - SF=1 or ZF=1
- jl Dest  \[ \text{PC} \leftarrow \text{Dest} \text{ iff last result} < 0 \]
  - SF=1 and Z=0
- je Dest  \[ \text{PC} \leftarrow \text{Dest} \text{ iff last result} = 0 \]
  - ZF=1
- jne Dest  \[ \text{PC} \leftarrow \text{Dest} \text{ iff last result} \neq 0 \]
  - ZF=0
- jge Dest  \[ \text{PC} \leftarrow \text{Dest} \text{ iff last result} \geq 0 \]
  - SF=0 or ZF=1
- jg Dest  \[ \text{PC} \leftarrow \text{Dest} \text{ iff last result} > 0 \]
  - SF=0 and Z=0

If the last result is not what is specified, then the jump is not taken; and the next sequential instruction is executed i.e. \[ \text{PC} = \text{PC} + \text{jump instruction size} \text{ vice Dest} \]
**Y86 example program w/ loop**

# y86loop.ys
.pos 0x0

irmovl $0,%eax  # sum = 0
irmovl $1,%ecx  # num = 1
Loop:
addl %ecx,%eax  # sum += num
irmovl $1,%edx  # tmp = 1
addl %edx,%ecx  # num++
irmovl $1000,%edx  # lim = 1000
subl %ecx,%edx  # if lim - num >= 0
jge Loop  # loop again
halt

Which instructions set the CC bits?

`CONVERT TO C`

```c
sum=0;
num = 1;
do {
  sum += num;
  num++;
}
while (1000 - num >= 0)
```

What are the flags set to for each instruction?
% yis y86loop.yo
Stopped in 6003 steps at PC = 0x23.
Status 'HLT', CC Z=0 S=1 O=0

Changes to registers:
%eax:  0x00000000  0x0007a314
%ecx:  0x00000000  0x000003e9
%edx:  0x00000000  0xffffffff # all one’s = -1

EXPLANATION:
1. Sum of 1 to 1000 = 500500 = 0x7a314 = %eax
2. 0x3e9 = 1001 = %ecx
3. CC sign bit = 1 because 1000 – 1001 = -1 = 0xffffffff = %edx
Conditional move

- **Move Unconditionally**
  - `rrmovl rA, rB`  
  - Encodings differ only by “function code”

- **Move When Less or Equal**
  - `cmovle rA, rB`  
  - Based on values of condition codes

- **Move When Less**
  - `cmovl rA, rB`  
  - Variants of `rrmovl` instruction

- **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”
Conditional move examples

```
# y86ccmov.ys
.pos 0x0
irmovl $1, %eax
cmpeq %eax,%ecx
irmovl 0, %ebx
addl %eax, %eax
cmpeqg %eax, %ebx
andl %ebx, %ebx
subl %eax, %ecx
cmpeqg %ecx, %edx
irmovl $0x7fffffff, %edx
addl %edx, %edx
halt
```

The cmovxx statement only moves the source register value to the destination register if “the condition is true”, so:

If the condition is “equal”, that means the CC bits have the ZF set to 1 i.e. the previous result was equal to zero,

cmpeq – checks if the previous result was greater than zero (i.e. SF=0 and Z=0) and if so, moves the source register value to the destination register

ETC
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
  - When pushing, must first decrement stack pointer then store
  - When popping, retrieve value then increment stack pointer

Better representation: right side
Stack Operations

Stack for Y86 works just the same as with IA32

- **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

R[%esp] ← R[%esp] - 4
M[R[%esp]] ← R[rA]

Increasing Addresses

Stack:

- `%esp`:
  - `pushl rA`:
    - `%esp` → `%esp - 4`
    - `rA`:
      - `pushl rA`:
        - `%esp` → `%esp - 4`

Value:

- `popl rA`
  - `rA` ← `value`
### Subroutine call and return

<table>
<thead>
<tr>
<th>call Dest</th>
<th>8</th>
<th>0</th>
<th>Dest</th>
</tr>
</thead>
</table>

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

**Note:** call uses absolute addressing

<table>
<thead>
<tr>
<th>ret</th>
<th>9</th>
<th>0</th>
</tr>
</thead>
</table>

- Pop value from stack
- Use as address for next instruction
- Like IA32
Procedure calls and return support

- Call pushes the return value onto the top of the stack
  - Dest R[%esp] ← R[%esp] - 4
    - Make space on the stack
  - M[R[%esp]] ← PC
    - Move the value of the PC, which has been incremented to the next instruction, and store it in the memory location pointed to by reg %esp
  - PC ← Dest
    - Move the destination address of the routine being called into the PC

- ret
  - PC ← M[R[%esp]]
    - Get the return address off the stack
  - R[%esp] ← R[%esp] + 4
    - Adjust the stack pointer
The stack

Initially

<table>
<thead>
<tr>
<th>%eax</th>
<th>0x123</th>
</tr>
</thead>
<tbody>
<tr>
<td>%edx</td>
<td>0</td>
</tr>
<tr>
<td>%esp</td>
<td>0x108</td>
</tr>
</tbody>
</table>

pushl %eax

<table>
<thead>
<tr>
<th>%eax</th>
<th>0x123</th>
</tr>
</thead>
<tbody>
<tr>
<td>%edx</td>
<td>0</td>
</tr>
<tr>
<td>%esp</td>
<td>0x104</td>
</tr>
</tbody>
</table>

popl %edx

<table>
<thead>
<tr>
<th>%eax</th>
<th>0x123</th>
</tr>
</thead>
<tbody>
<tr>
<td>%edx</td>
<td>0x123</td>
</tr>
<tr>
<td>%esp</td>
<td>0x108</td>
</tr>
</tbody>
</table>

 irmovl $4, %ecx
 subl %ecx, %esp
 movl %eax, (%esp)

 movl (%esp), %eax
 irmovl $4, %ecx
 addl %ecx, %esp
```c
#include <stdio.h>

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

void main() {
    int c = 4;
    int a[4] = {5, 6, 7, 8};
    int x = Sum(a, c);
    // printf("x = %d", x);
}
```

Swap example

NOTE: IA32 move statements

```c
void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
} //codeswap.c

REMINDER: we use pointers so can pass address since can’t pass values back outside of the function... push right to left
```
Understanding Swap
Notice decreasing address on stack

1. Move 0x124 to %edx
2. Move 0x120 to %ecx
3. Move 123 to %ebx
4. Move 456 to %eax
5. Move 456 to M[0x124]
6. Move 123 to M[0x120]

pushl %ebp
movl %esp, %ebp
pushl %ebx
movl 8(%ebp), %edx  # edx = xp
movl 12(%ebp), %ecx  # ecx = yp
movl (%edx), %ebx  # ebx = *xp (t0)
movl (%ecx), %eax  # eax = *yp (t1)
movl %eax, (%edx)  # *xp = t1
movl %ebx, (%ecx)  # *yp = t0

Top of stack

popl %ebx
popl %ebp
ret
Stack frames for caller and swap_add

Fig 3.24
The machine uses the stack to:
- Pass procedure arguments
- Store return information
- Save registers for later restoration
- Local storage

**Stack frame**

- Portion of the stack allocated for a single procedure call
- The topmost stack frame is delimited by two pointers:
  - Register %ebp – the frame/base ("bottom") pointer
  - Register %esp – the stack pointer ("top")
    - Can move while the procedure is executing HENCE
    - MOST INFORMATION IS ACCESSED RELATIVE TO THE FRAME/BASE POINTER
    - Indicates lowest stack address i.e. address of top element
Procedure calls (cont)

- Procedure P (the “caller”) calls procedure Q (the “callee”)
- Caller stack frame (P)
  - The arguments to Q are contained within the stack frame for P
  - The first argument is always positioned at offset 8 relative to %ebp
  - Remaining arguments stored in successive bytes (typically 4 bytes each but not always)... +4+4n is return address plus 4 bytes for each argument.
  - When P calls Q, the return address within P where the program should resume execution when it returns from Q is pushed on to the stack
- Callee stack frame (Q)
  - Saved value of the frame pointer
  - Copies of other saved registers
  - Local variables that cannot all be stored in registers (see next slide)
  - Stores arguments to any procedures it calls.

```
int P(int x) {
    int y=x*x;
    int z=Q(y);
    return y+z;
}
```
You can allocate space on the stack by decrementing the stack pointer by an appropriate amount (push).

Space can be deallocated by incrementing the stack pointer (pop).

Local variables on the stack:
- There are a limited number of registers.
- Stack is used to store local variables some of which can be arrays or structures.
- When using `&` applied to a local variable, there must be an address than can be generated for it.
Procedure call and return

- **Call instruction**
  - Has a label which is a target indicating the address of the instruction where the called procedure (the callee) starts
  - Direct or indirect label
  - Push a return address on the stack
    - Is the address of the instruction immediately following the call in the (assembly) program
  - Jump to the start of the called procedure

- **Return instruction**
  - Pops an address off the stack
  - Jumps to this location
  - FYI: proper use is to have prepared the stack so that the stack pointer points to the place where the preceding call instruction stored its return address

- **Leave instruction** is equivalent to:
  - `movl %ebp, %esp`
  - `popl %ebp`

---

**Reminder of call:**
Dest R[%esp]←R[%esp]-4
M[R[%esp]]←PC
PC←Dest

**Reminder of return:**
PC←M[R[%esp]]
R[%esp]←R[%esp]+4
Procedure call and return

// Beginning of function sum
08048394  <sum>:
  8048394:  55  push %ebp
  ...
//return from function sum
  80493a4:  c3  ret
  ...
// call to sum from main  - START HERE!
  80483dc:  e8 b3 ff ff ff  call  8048394 <sum>
  80483e1:  83 c4 14  add $0x14,%esp

Return address
Register usage conventions

- Program registers are a shared resource
- One procedure is active at a given time
- Don’t want the callee to overwrite a value the caller planned to use later
- BY CONVENTION/PROTOCOL
  - “Caller-save” registers: %eax, %edx and %ecx
    - When Q is called by P, it can overwrite these registers without destroying any data required by P
  - “Callee-save” registers: %ebx, %esi and %edi
    - Q must save these values on the stack before overwriting them, and restore them before returning
  - %ebp and %esp must be maintained
  - Register %eax is used for returning the value from any function that returns an integer or pointer.

int P(int x)
{
    int y=x*x;
    int z=Q(y);
    return y+z;
}

1. The caller, P, can save the value y.
2. P can store the value in a callee-save register (saved and restored).
Miscellaneous instructions

Why need a nop? See next slide

- Don’t do anything

- 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
Why need a NOP instruction?

Pipelining Issues

<table>
<thead>
<tr>
<th>Fetch</th>
<th>Execute</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Fetch</td>
</tr>
<tr>
<td></td>
<td>Execute</td>
</tr>
<tr>
<td></td>
<td>Fetch</td>
</tr>
<tr>
<td></td>
<td>Execute</td>
</tr>
</tbody>
</table>

**EX.1 First instruction is a call or unconditional Jump**

- What instruction is being executed next?
  - Should be Dest, but fetches next instruction
  - Needs a NOP instruction

**Ex.2 First instruction is a conditional Jump**

- What instruction is being executed next?
  - Either jump or drop to next instruction; but fetches next instruction
  - Needs a NOP instruction
Y86 Instruction Set

- Encoding of each instruction
- SEQ Hardware Structure
- SEE HANDOUT

Recall Memory is a Bunch of Bits!
How do we know if it is an instruction or not?
How do we know which instruction, which operands, etc.

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

Instruction Format

- Instruction byte: i
- Optional register byte: rA:rB
- Optional constant word: valC
SEQ Hardware Structure
Abstract and Stages

**State**
- Program counter register (PC)
- Condition code register (CC)
- Register File
- Memories
  - Access same memory space
  - Data: for reading/writing program data
  - Instruction: for reading instructions

**Instruction Flow**
- Read instruction at address specified by PC
- Process through stages
- Update program counter

**Fetch**
- Read instruction from instruction memory
- If PC points to it, we view it as instruction

**Decode**
- Read program registers

**Execute**
- Compute value or address

**Memory**
- Read or write data

**Write Back**
- Write program registers

**PC**
- Update program counter
Executing \( \rightarrow \) rmmovl

- Fetch
  - Read 6 bytes
- Decode
  - Read operand registers
- Execute
  - Compute effective address
- Memory
  - Write to memory
- Write back
  - Do nothing
- PC Update
  - Increment PC by 6

- Use ALU for address computation
Executing → Arithmetic/Logical Ops

- Fetch
  - Read 2 bytes
- Decode
  - Read operand registers
- Execute
  - Perform operation
  - Set condition codes
- Memory
  - Do nothing
- Write back
  - Update register
- PC Update
  - Increment PC by 2

Formulate instruction execution as sequence of simple steps
Use same general form for all instructions; often called Register Transfer Language (RTL)
Executing → popl

- **Fetch**
  - Read 2 bytes

- **Decode**
  - Read stack pointer

- **Execute**
  - Increment stack pointer by 4

- **Memory**
  - Read from old stack pointer

- **Write back**
  - Update stack pointer
  - Write result to register

- **PC Update**
  - Increment PC by 2

---

**popl rA**

<table>
<thead>
<tr>
<th>Fetch</th>
<th></th>
<th></th>
<th>Read instruction byte</th>
<th>Read register byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>iCode:ifun ← M₁[PC]</td>
<td>rA:rB ← M₁[PC+1]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>valP ← PC+2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decode</th>
<th></th>
<th></th>
<th>Compute next PC</th>
<th>Read stack pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>valA ← R[%esp]</td>
<td>valB ← R[%esp]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Execute</th>
<th></th>
<th></th>
<th>Read from stack</th>
<th>Update stack pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>valE ← valB + 4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory</th>
<th></th>
<th></th>
<th>Read from stack</th>
<th>Update stack pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>valM ← M₄[valA]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Write back</th>
<th></th>
<th></th>
<th>Write back result</th>
<th>Update PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>R[%esp] ← valE</td>
<td>R[rA] ← valM</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PC update</th>
<th></th>
<th></th>
<th>Update PC</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PC ← valP</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Executing → Jumps

- Fetch
  - Read 5 bytes
  - Increment PC by 5
- Decode
  - Do nothing
- Execute
  - Determine whether to take branch based on jump condition and condition codes
- Memory
  - Do nothing
- Write back
  - Do nothing
- PC Update
  - Set PC to Dest if branch taken or to incremented PC if not branch
**Executing → Call**

- **Fetch**
  - Read 5 bytes
  - Increment PC by 5
- **Decode**
  - Read stack pointer
- **Execute**
  - Decrement stack pointer by 4
- **Memory**
  - Write incremented PC to new value of stack pointer
- **Write back**
  - Update stack pointer
- **PC Update**
  - Set PC to Dest

**Diagram:**

- **Fetch**
  - `icode:ifun ← M_t[PC]`
  - `valC ← M_t[PC+1]`
  - `valP ← PC+5`

- **Decode**
  - `valB ← R[esp]`

- **Execute**
  - `valE ← valB + 4`

- **Memory**
  - `M_4[valE] ← valP`

- **Write back**
  - `R[esp] ← valE`

- **PC update**
  - `PC ← valC`

- **Use ALU to decrement stack pointer**
- **Store Incremented PC**

**Table:**

- **call Dest**
  - `Dest`
  - `return: xx xx`
  - `target: xx xx`
Executing → ret

- Fetch
  - Read 5 bytes
  - Increment PC by 5
- Decode
  - Read stack pointer
- Execute
  - Decrement stack pointer by 4
- Memory
  - Write incremented PC to new value of stack pointer
- Write back
  - Update stack pointer
- PC Update
  - Set PC to Dest

Use ALU to increment stack pointer
Read return address from memory
We’ve already done the instruction encoding practice, but next are some good examples from the book
Determine the byte encoding of the following x86 instruction sequence given ".pos 0x100" specifies the starting address of the object code to be 0x100 (practice problem 4.1)

```
.pos 0x100 # start code at address 0x100
irmovl   $15, %ebx       # load 15 into %ebx
rrmovl  %ebx, %ecx      # copy 15 to %ecx
loop:
    rmmovl  %ecx, -3(%ebx) # save %ecx at addr 15-3=12
    addl   %ebx, %ecx      # increment %ecx by 15
    jmp    loop            # goto loop
```
**Instruction encoding practice (cont)**

- **0x100:** 30f3fcfffffff 406300080000 00
  - 0x100: 30f3fcfffffff  
    - irmovl $-4, %ebx
  - 0x106: 406300080000  
    - rmmovl %esi, 0x800(%ebx)
  - 0x10c: 00  
    - halt

Now you try:

- **0x200:** a06f80080200000030f30a00000090
- **0x400:** 6113730004000000
Summary

Important property of any instruction set

THE BYTE ENCODINGS MUST HAVE A UNIQUE INTERPRETATION

which

ENSURES THAT A PROCESSOR CAN EXECUTE AN OBJECT-CODE PROGRAM WITHOUT ANY AMBIGUITY ABOUT THE MEANING OF THE CODE