Assembly

- 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)
ISA – 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
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
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
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????????

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

```
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
```
Y86 Simulator

- Contents of memory
- Processor State
  - The fetch-execute loop
- Register file
- Status
- Condition Codes
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></td>
</tr>
<tr>
<td>%ecx</td>
<td>%edi</td>
<td></td>
</tr>
<tr>
<td>%edx</td>
<td>%esp</td>
<td></td>
</tr>
<tr>
<td>%ebx</td>
<td>%ebp</td>
<td></td>
</tr>
</tbody>
</table>

ZF | SF | OF
PC
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

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>AOK</td>
<td>1</td>
</tr>
<tr>
<td>HLT</td>
<td>2</td>
</tr>
<tr>
<td>ADR</td>
<td>3</td>
</tr>
<tr>
<td>INS</td>
<td>4</td>
</tr>
</tbody>
</table>

- 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
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
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 → “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)
Supported OPs and Jump

- **OP1 (opcode = 6)**
  - Only take registers as operands
  - Only work on 32 bits
  - Note: no “or” and “not” ops
  - **Only instructions to set CC**

- **Jump instructions (opcode = 7)**
  - fn = 0 for unconditional jump
  - fn = 1-6 for <= < = != >= >
  - Refer to generically as “j XX”
  - 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

<table>
<thead>
<tr>
<th>fn</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>addl</td>
</tr>
<tr>
<td>1</td>
<td>subl</td>
</tr>
<tr>
<td>2</td>
<td>andl</td>
</tr>
<tr>
<td>3</td>
<td>xorl</td>
</tr>
</tbody>
</table>
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
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
## Move operation (cont)

<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
- **mrmmovl** loads a word from memory
- **rmmovl** and **mrmmovl** are the only instructions that access memory - Y86 is a load/store architecture
## Conditional move

### Move Unconditionally
- `rrmovl rA, rB` 2 0 rA rB

### Move When Less or Equal
- `cmovle rA, rB` 2 1 rA rB

### Move When Less
- `cmovl rA, rB` 2 2 rA rB

### Move When Equal
- `cmovne rA, rB` 2 3 rA rB

### Move When Not Equal
- `cmovnle rA, rB` 2 4 rA rB

### Move When Greater or Equal
- `cmovge rA, rB` 2 5 rA rB

### Move When Greater
- `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 `rrmovl` instruction
  - (conditionally) copy value from source to destination register
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
  - When popping, increment stack pointer

Increasing Addresses

Increasing Addresses
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

Stack for Y86 works just the same as with IA32
Subroutine call and return

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

Note: call uses absolute addressing

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

- **nop**
  1 0
  - Don’t do anything

- **halt**
  0 0
  - 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
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: icode:ifun
- Optional register byte: rA:rB
- Optional constant word: valC
SEQ Hardware Structure
Abstract and Stages

<table>
<thead>
<tr>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>■ Program counter register (PC)</td>
</tr>
<tr>
<td>■ Condition code register (CC)</td>
</tr>
<tr>
<td>■ Register File</td>
</tr>
<tr>
<td>■ Memories</td>
</tr>
<tr>
<td>✦ Access same memory space</td>
</tr>
<tr>
<td>✦ Data: for reading/writing program data</td>
</tr>
<tr>
<td>✦ Instruction: for reading instructions</td>
</tr>
</tbody>
</table>

**Instruction Flow**

<table>
<thead>
<tr>
<th>Fetch</th>
</tr>
</thead>
<tbody>
<tr>
<td>■ Read instruction from instruction memory</td>
</tr>
<tr>
<td>■ If PC points to it, we view it as instruction</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decode</th>
</tr>
</thead>
<tbody>
<tr>
<td>■ Read program registers</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Execute</th>
</tr>
</thead>
<tbody>
<tr>
<td>■ Compute value or address</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>■ Read or write data</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Write Back</th>
</tr>
</thead>
<tbody>
<tr>
<td>■ Write program registers</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>■ Update program counter</td>
</tr>
</tbody>
</table>
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

---

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

<table>
<thead>
<tr>
<th>OPI rA, rB</th>
<th>Decode</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>valA ← R[rA]</td>
</tr>
<tr>
<td></td>
<td>valB ← R[rB]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OPI rA, rB</th>
<th>Execute</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>valE ← valB OP valA</td>
</tr>
<tr>
<td></td>
<td>Set CC</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OPI rA, rB</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Write back</td>
</tr>
<tr>
<td></td>
<td>R[rB] ← valE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OPI rA, rB</th>
<th>Write back</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PC ← valP</td>
</tr>
</tbody>
</table>

- **Formulate instruction execution as sequence of simple steps**
- **Use same general form for all instructions; often called Register Transfer Language (RTL)**
Executing \( \rightarrow \text{rmmovl} \)

- **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**

<table>
<thead>
<tr>
<th><strong>Fetch</strong></th>
<th>rmmovl rA, D(rB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>iCode: ifun ( \leftarrow ) M_1[PC]</td>
<td></td>
</tr>
<tr>
<td>rA:rB ( \leftarrow ) M_1[PC+1]</td>
<td></td>
</tr>
<tr>
<td>valC ( \leftarrow ) M_4[PC+2]</td>
<td></td>
</tr>
<tr>
<td>valP ( \leftarrow ) PC+6</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Decode</strong></th>
<th>valA ( \leftarrow ) R[rA]</th>
</tr>
</thead>
<tbody>
<tr>
<td>valB ( \leftarrow ) R[rB]</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Execute</strong></th>
<th>valE ( \leftarrow ) valB + valC</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>Memory</strong></th>
<th>valA ( \leftarrow ) R[rA]</th>
</tr>
</thead>
<tbody>
<tr>
<td>M_4[valE] ( \leftarrow ) valA</td>
<td></td>
</tr>
</tbody>
</table>

| **Write back** | |
|----------------| |

| **PC update** | |
|---------------| |
| PC \( \leftarrow \) valP |

**Read instruction byte**
**Read register byte**
**Read displacement D**
**Compute next PC**
**Read operand A**
**Read operand B**
**Compute effective address**
**Write value to memory**
**Update PC**
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

---

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

<table>
<thead>
<tr>
<th>Decode</th>
<th>valA ← R[%esp]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>valB ← R [%esp]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Execute</th>
<th>valE ← valB + 4</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Memory</th>
<th>valM ← M₄[valA]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write</td>
<td>R[%esp] ← valE</td>
</tr>
<tr>
<td>back</td>
<td>R[rA] ← valM</td>
</tr>
</tbody>
</table>

| PC update      | PC ← valP       |

- **Use ALU to increment stack pointer**
- **Must update two registers**
  - Popped value
  - New stack pointer
**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[PC]`
- **Decode**:
  - `valC ← M[PC+1]`
  - `valP ← PC+5`
- **Execute**:
  - `valE ← valB + 4`
- **Memory**:
  - `M[4][valE] ← valP`
- **Write back**:
  - `R[jesp] ← valE`
- **PC update**:
  - `PC ← valC`

**Call 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
Instruction encoding practice

Determine the byte encoding of the following Y86 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: 30f3fcffffff 406300080000 00
0x100: 30f3fcffffff
0x106: 406300080000
0x10c: 00

irmovl $-4, %ebx
rmmovl %esi, 0x800(%ebx)
halt

Now you try:

0x200: a06f80080200000030f30a00000090
0x400: 6113730004000000
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