

CSCE 312 LAB 5

ZACH584

Section 504 - 2:35PM - 3:25PM

Section 505 - 3:55PM - 4:45PM

Section 506 - 5:30PM - 6:20PM

Section 507 - 6:40PM - 7:30PM

If you have not yet shown the demonstration for Lab4, Tuesday is the deadline (without penalty) and Thursday(with penalty).


LAB5

- Report Deadline: 4th April midnight (We will start Lab 6 on 4th if possible)
- To be done individually

SETUP for LAB5

Chapter 3: Machine-Level Representation of Programs

Chapter 4: Processor Architecture

- Y86-64 tools and documentation
 - Source distribution (README) 
 - Simulator guide (pdf)
 - Technical report describing a successful effort at formally verifying pipelined Y86 implementations. (pdf)

- Follow this link to download and decompress the source code:
 - <http://csapp.cs.cmu.edu/3e/students.html>
- Use a linux Terminal, locate the decompressed “sim” folder.
- Comment the two lines `TKLIBS=-L...` and `TKINC=-isystem...` in Makefile inside sim folder
- Use command `make clean; make` to cleanup existing binaries and recompile the code.
- Notes:
 - If you don't have a linux machine, please use the `compute.cse.tamu.edu` server
 - If you use any linux machine other than the school server, you may encounter some library missing errors. Please search and install all dependencies then try recompiling the code.

SETUP for LAB5

- Once the compilation is complete, check for the binaries of YAS and YIS in the `misc` folder.

- If they're there, then you're good to go.

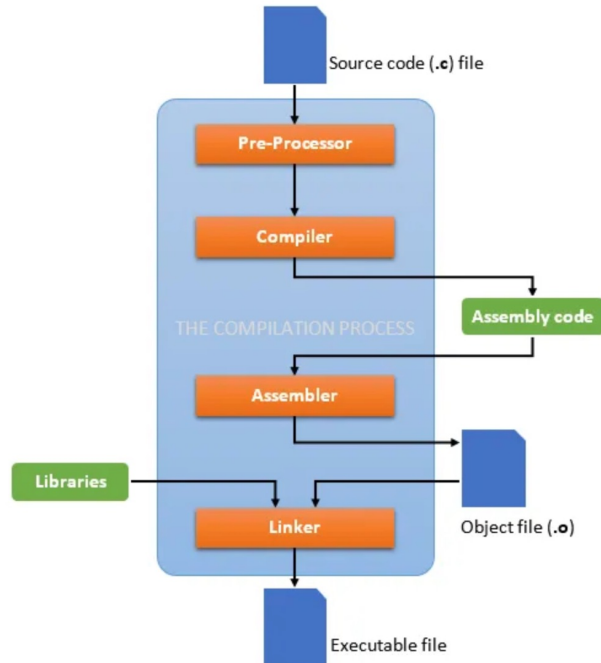
- To run the `asum.y` example, type two commands:

```
./misc/yas y86-code/asum.y ./misc/yis y86-code/asum.yo
```

- The first command converts the Y86-64 assembly code (.y) into object machine code (.yo)

- The 2nd command simulates the instructions and prints out the changes in the state of registers and memory.

Steps in Program Execution



- Pre processing: gets rid of comments, replaces macros, includes header files
- Compiler: Parses through the code and converts the high level code to an Intermediate Representation [Assembly Code]
- Assembler: Takes this Assembly code and converts it into binary
- Linker: links multiple such binaries so that functions from other binaries can be used and finally an executable file is created

Y86-64 [Assembly Code]

- Instructure Set Architecture similar and much simpler form of x86
- PROGRAM COUNTER – indicates address of current instruction that is being implemented
- REGISTERS - 64bit meaning each register is of length 64bits
- MEMORY (DRAM) – Each address corresponds to 64bit
- CONDITION FLAGS – 1 bit flags set by arithmetic and logical operations
 - OF : Overflow
 - ZF : Zero
 - SF : Negative
- STATUS REGISTER - indicates state of current execution
 - AOK : Normal Operation
 - HLT : Program halted
 - INS : Invalid instruction encountered
 - ADR : Bad address encountered

REGISTERS

- Each register has 4-bit ID

<code>%rax</code>	0	<code>%r8</code>	8
<code>%rcx</code>	1	<code>%r9</code>	9
<code>%rdx</code>	2	<code>%r10</code>	A
<code>%rbx</code>	3	<code>%r11</code>	B
<code>%rsp</code>	4	<code>%r12</code>	C
<code>%rbp</code>	5	<code>%r13</code>	D
<code>%rsi</code>	6	<code>%r14</code>	E
<code>%rdi</code>	7		F

Y86-64 Instruction Set - Operations

<code>addq rA, rB</code>
<code>subq rA, rB</code>
<code>andq rA, rB</code>
<code>xorq rA, rB</code>

Let `rdx = 5, rax = 2`

- `addq rA rB` - add two register value ($\text{Reg}[\text{rB}] = \text{Reg}[\text{rB}] + \text{Reg}[\text{rA}]$)
 - `addq %rdx, %rax`
 - result : `rax : 7, rdx : 5`
- `subq rA rB` - subtract two register value ($\text{Reg}[\text{rB}] = \text{Reg}[\text{rB}] - \text{Reg}[\text{rA}]$)
 - `subq %rax, %rdx`
 - result : `rdx : 3, rax : 2`

Let `rdx = 6, rax = 4 (110, 100)`

- `andq rA rB` - bitwise and operation, $\text{Reg}[\text{rB}] = \text{Reg}[\text{rB}] \& \text{Reg}[\text{rA}]$
 - `andq %rdx, %rax`
 - Result : `rax : 4, rdx : 6`
- `xorq rA rB`: bitwise or operation, $\text{Reg}[\text{rB}] = \text{Reg}[\text{rB}] \wedge \text{Reg}[\text{rA}]$
 - `xorq %rdx, %rax`
 - result : `rax : 2, rdx : 6`

Y86-64 Instruction Set -

<code>jmp L</code>
<code>jle L</code>
<code>jl L</code>
<code>je L</code>
<code>jne L</code>
<code>jge L</code>
<code>jg L</code>

- `jmp` - jump to that destination
- `jle` - if less or equal , jump to Dest if last result ≤ 0
- `jl` - if less, jump to Dest if last result < 0
- `je` - if equal, jump to Dest if last result $= 0$
- `jne` - if not equal, jump to Dest if last result $\neq 0$
- `jge` - if greater or equal, jump to Dest if last result ≥ 0
- `jg` - if greater, jump to Dest if last result > 0

Y86-64 Mov instructions

Instruction	Source	Destination
<code>rrmovq rA, rB</code>	Register	Register
<code>irmovq V, rB</code>	Immediate	Register
<code>rmmovq rA, D(rB)</code>	Register	Memory
<code>mrmovq D(rA), rB</code>	Memory	Register

- `irmovq` : immediate value (number) into register
 - `irmovq $3, %rax` , will store value 3 to the rax register
- `cmovq` : move one register value to another
 - `cmovq %rax, %rcx` , now rcx has value 3
- `rmmovq` : move register value to memory
 - `rmmovq %rax, 0(%rbx)` ; assume rbx : 0x4000, now at address 0x4000, it contains 3
 - `rmmovq %rax, 4(%rbx)` ; number in front : offset, add 4 to that memory address, 0x4004 contains 3
- `mrmovq` : move value from memory to a register
 - `mrmovq 4(%rbx), %rdx`; move value at 0x4004 into rdx, which is 3 now.

PROBLEM 1, 2

- Write the correct y86-64 assembly code in .ys files for the C code given in the question
- Generate .yo files from .ys files:
 - `> ./yas <filename>.ys`
- Execute the code from the .yo file:
 - `> ./yis <filename>.yo`
- After successful execution it shows you the changes in registers and memory.
- Helpful Resource:
 - https://csit.kutztown.edu/~schwesin/fall21/csc235/lectures/Instruction_Set_Architecture.html

PROBLEM 3,4,5

- Generate x86 assembly code in .s files from the C code given to you and analyse the assembly code as instructed in the questions
 - `> gcc -S <filename>.c -o <filename>.s`
 - .s file has the assembly code
- Helpful Resource:
 - <https://medium.com/@laura.derohan/compiling-c-files-with-gcc-step-by-step-8e78318052>

PROBLEM 6

- Use inline x86 assembly to add assembly code to a C program.
- You have to use a linux machine to solve this problem. If you donot have a linux, then use the CSE Linux server that you used in Lab1
- How to do that?
 - <https://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html>
 - [Resource 1](#)
 - [Resource 2](#)
 - [Resource 3](#)
 - [Resource 4](#)

Files to be submitted

Report with screenshots of :

- memory and register changes for Problem 1 and Problem 2
- comparison of assembly code for Problems 3,4,5
- output received on running code with inline assembly

yas executable

ylis executable

.yo and .ys files for Problem 1 and Problem 2

.s files for Problem 3,4,5

.c file for Problem 6