


# Problem 7: Some Assembly Required (25 points) [Email Submission]

By Henry Tang and Darius Jankauskas

The future is now! Chris Goober, founder of the highly successful tech company *Binary Bandits Inc.* and the world's leading programming languages expert, has developed a brand new assembly language that far exceeds the capabilities of both x86 and ARM.[Citation needed.] Not only does this new language result in faster performance, it is also more power efficient and provides better memory safety, all while being packaged in a simple and elegant instruction set. It has been fittingly named *AWE*, which stands for *Assembly with Epicness*.

Chris plans to unveil *AWE* to the public next month at his company's annual developer conference. Even though the language is complete, Chris hopes to augment his grand event by also unveiling a sophisticated profiler for *AWE*. To particularly impress his company's shareholders, he insists on implementing the profiler in *AWE* as well. However, he is a bit time crunched and has delegated the implementations of certain functionalities to you. Can you implement these features and impress the legendary Chris Goober?



PRINCETON Computer Science
Keller Center for Innovation in Engineering Education
CITADEL | CITADEL Securities
OLD MISSION
RADIX
PACM
PRINCETON GEOSCIENCES
SIG SUSQUEHANNA INTERNATIONAL GROUP, LLP
Evergreen Statistical Trading
PICSciE
ORFE Operations Research & Financial Engineering
Department of Mathematics
CENTER FOR STATISTICS AND MACHINE LEARNING
Electrical and Computer Engineering
MECHANICAL & AEROSPACE ENGINEERING
Princeton Physics




## Background: Assembly Code

*This is a brief overview of assembly languages. If you already know this, feel free to skip to the instruction set section.*

High level programming languages such as Python, Java, or even C cannot be directly interpreted and run on a computer. Instead, a compiler must first compile these programs down into bytecode or machine code before they can be executed by an interpreter or processor.

Consider the example of C. During the compilation of C down to machine code (which consists of just a bunch of 0s and 1s), it is first compiled into an intermediate state known as assembly code. Depending on the computer, this will typically be written in either the x86 or the ARM assembly language. Assembly code is readable and writeable by humans, but only just and with great difficulty. One should hope they never have to spend too much time coding in assembly for their own sanity.

Each line of an assembly program contains a single instruction, and starts with a short mnemonic that indicates the type of instruction to be executed. Examples of mnemonics include *add* to add two numbers, and *j* to jump to another part of the code. These mnemonics are then followed by a series of parameters for the instruction. During execution, assembly code uses a structure known as a program counter (pc) to identify which line should be executing at a given moment,. The value stored in the pc is the next instruction to execute, and the pc increments after said instruction is executed. The only time the pc does not increment by a fixed predictable amount is when it executes a jump instruction or a branch instruction that is taken.

Assembly code does not use variables like high level programming languages. Instead, there are a fixed set of registers that can store and manipulate values. In addition, they do not contain constructs such as if statements and loops. Instead, in order to handle these kinds of code behavior, assembly code uses branch and jump instructions.



PRINCETON Computer Science
Keller Center for Innovation in Engineering Education
CITADEL | CITADEL Securities
OLD MISSION
RADIX
PACM
PRINCETON GEOSCIENCES
SIG SUSQUEHANNA INTERNATIONAL GROUP, LLP
Evergreen Statistical Trading
PICSciE
ORFE Operations Research & Financial Engineering
Department of Mathematics
CENTER FOR STATISTICS AND MACHINE LEARNING
Electrical and Computer Engineering
MECHANICAL & AEROSPACE ENGINEERING
Princeton Physics




## Instruction Set

In *AWE*, there are 16 registers labelled r0 to r15. All registers store 64 bit signed integers. There is word-addressable instruction memory numbered from 0 to 1023. All instructions are 1 word long. There is no data memory. All registers may be set to any 64 bit signed value.

## Simulator

We have provided a simulator for executing *AWE* assembly here. Note that the simulator is whitespace insensitive. Each successive instruction is stored at the next word address. The first instruction is stored at word 0.

You will notice there are two text boxes in the simulator. The first box inserts instructions into addresses 0 to 99, while the second box inserts instructions starting from address 100. Their use will become apparent as you read through the remainder of the problem statement. For now, just know that if you want to experiment with *AWE*, you can do so by writing the code into the first text box.

The simulator also adds comments to the assembly language. The characters on a line after the character ; are ignored by the simulator.

If the simulator freezes after you click run, then there is probably an infinite loop in your code. You can either wait for your browser to throw an error, or close the tab and reopen it again.



PRINCETON Computer Science
Keller Center for Innovation in Engineering Education
CITADEL | CITADEL Securities
OLD MISSION
RADIX
PACM
PRINCETON GEOSCIENCES
SIG SUSQUEHANNA INTERNATIONAL GROUP, LLP
Evergreen Statistical Trading
PICSciE
ORFE Operations Research & Financial Engineering
Department of Mathematics
CENTER FOR STATISTICS AND MACHINE LEARNING
Electrical and Computer Engineering
MECHANICAL & AEROSPACE ENGINEERING
Princeton Physics




| *AWE* Instruction Set | | |
|---|---|---|
| Instruction Mnemonic | Instruction Description | Example |
| nop | Has no effect | nop |
| mov reg, const | Moves 64 bit signed constant into reg | mov r0, 2 |
| add reg0, reg1, reg2 | Store reg1 + reg2 into reg0 | add r0, r1, r2 |
| sub reg0, reg1, reg2 | Store reg1 − reg2 into reg0 | sub r0, r1, r2 |
| mul reg0, reg1, reg2 | Store reg1 × reg2 into reg0 | mul r0, r1, r2 |
| div reg0, reg1, reg2 | Store reg1 / reg2 into reg0, where / represents integer division | div r0, r1, r2 |
| print | Prints assembler textual representation of previous instruction in memory, located at pc − 1 | print |
| printr | Prints previous instruction in memory, located at pc − 1, as well as the values of registers r0 through r15 | printr |
| copy reg0, reg1 | Copies instruction in instruction memory from the address in reg1 into the address in reg0 | copy r0, r1 |
| beq reg0, reg1, reg2 | If reg0 = reg1, then newpc ← reg2. Otherwise, newpc ← pc + 1. | beq r0, r1, r2 |
| bne reg0, reg1, reg2 | If reg0 ≠ reg1, then newpc ← reg2. Otherwise, newpc ← pc + 1. | bne r0, r1, r2 |
| j reg | newpc ← reg and r15 ← pc | j r0 |
| rj reg | newpc ← pc + reg | rj r5 |
| halt | terminate execution | halt |
| write reg, [inst] | Write inst into instruction memory at address reg. Note that inst **cannot** be another write instruction | write r0, [add r1, r2, r3] |

Table 1: pc refers to the address of the instruction before it is executed. newpc refers to next value of the program counter after said instruction is executed.



PRINCETON Computer Science
Keller Center for Innovation in Engineering Education
CITADEL | CITADEL Securities
OLD MISSION
RADIX
PACM
PRINCETON GEOSCIENCES
SIG SUSQUEHANNA INTERNATIONAL GROUP, LLP
Evergreen Statistical Trading
PICSciE
ORFE Operations Research & Financial Engineering
Department of Mathematics
CENTER FOR STATISTICS AND MACHINE LEARNING
Electrical and Computer Engineering
MECHANICAL & AEROSPACE ENGINEERING
Princeton Physics




At the start of execution, all registers are set to 0 and pc = 0. In addition, any part of the instruction memory that hasn't been explicitly assigned an instruction is unspecified (meaning any value could be stored there). Note that you are allowed to overwrite unspecified locations in memory, but it would not be wise to access these locations before they have been overwritten.

As an example, consider the following code written in *AWE*.

```
mov r0, 3          ; line 0
mov r1, 6          ; 1
add r0, r0, r0     ; 2
mov r2, 6          ; 3
beq r0, r1, r2     ; 4
print              ; 5
print              ; 6
halt               ; 7
```

Running this will output

```
print
```

To see why, let's run through the code.

Initially, we load 3 and 6 into r0 and r1 respectively. Then, we perform the operation r0 ← r0 + r0, which results in r0 = 6. We load 6 into r2. Then, we have a branch equal instruction. Since r0 = r1, we branch to r2, which has value 6. In other words, we move pc to address 6, which contains the last print statement. The print instruction prints the previous instruction in memory, which in this case is the print instruction at address 5. Then, we encounter halt and exit the program.



PRINCETON Computer Science
Keller Center for Innovation in Engineering Education
CITADEL | CITADEL Securities
OLD MISSION
RADIX
SIG SUSQUEHANNA INTERNATIONAL GROUP, LLP
Evergreen Statistical Trading
PICSciE
ORFE Operations Research & Financial Engineering
Department of Mathematics
CENTER FOR STATISTICS AND MACHINE LEARNING
Electrical and Computer Engineering
MECHANICAL & AEROSPACE ENGINEERING
Princeton Physics




## Part 1 Static Analysis (10 points)

Your first goal is to create a static code tracer. In other words, given a program $\mathcal{P}$ written in *AWE*, you must write assembly code in *AWE* such that when your code run, it will print out the instructions of $\mathcal{P}$ in correct order.

### Input

You are provided a program $\mathcal{P}$ written in *AWE* that's 25 lines long. There will only be one halt instruction in $\mathcal{P}$ and it is 25th instruction. Your task is to write some assembly code $\mathcal{C}$ up to 100 lines long between addresses 0 to 99 (inclusive) that will print the 25 lines of $\mathcal{P}$.

The way we actually execute your code is we will take $\mathcal{C}$ and if it is not 100 lines long, append unspecified instructions to it until it is 100 lines long. Denote this extended code $\overline{\mathcal{C}}$. Then we append the 25 lines of $\mathcal{P}$ directly after $\overline{\mathcal{C}}$, so that $\mathcal{P}$ is located at addresses 100 to 124 (inclusive). The remainder of instruction memory is unspecified. We execute code with pc initially set to 0.

To run this on the simulator located here, insert your code $\mathcal{C}$ into the User Code text box, and insert your code $\mathcal{P}$ into the second.

### Hint

To print, use the print assembly instruction! In addition, the write instruction may come in handy. printr is not required for this part, although it could be useful for debugging.

### Output

Your program should print 25 lines, corresponding to the 25 lines of $\mathcal{P}$.



PRINCETON Computer Science
Keller Center for Innovation in Engineering Education
CITADEL | CITADEL Securities
OLD MISSION
RADIX
PACM
PRINCETON GEOSCIENCES
SIG SUSQUEHANNA INTERNATIONAL GROUP, LLP
Evergreen Statistical Trading
PICSciE
ORFE Operations Research & Financial Engineering
Department of Mathematics
CENTER FOR STATISTICS AND MACHINE LEARNING
Electrical and Computer Engineering
MECHANICAL & AEROSPACE ENGINEERING
Princeton Physics




**Example**
Consider the following program $\mathcal{P}$.

```
mov r0, 124
beq r2, r3, r0
add r4, r0, r0
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
halt
```

Figure 1: One possible program $\mathcal{P}$ that we will provide.

The print output of your program $\mathcal{C}$ concatenated with $\mathcal{P}$ should be exactly the same as the 25 instructions above.



PRINCETON Computer Science
Keller Center for Innovation in Engineering Education
CITADEL | CITADEL Securities
OLD MISSION
RADIX
SIG SUSQUEHANNA INTERNATIONAL GROUP, LLP
Evergreen Statistical Trading
PICSciE
ORFE Operations Research & Financial Engineering
Department of Mathematics
CENTER FOR STATISTICS AND MACHINE LEARNING
Electrical and Computer Engineering
MECHANICAL & AEROSPACE ENGINEERING
Princeton Physics




## Part 2 Dynamic Analysis (15 points)

Your next goal is to create a dynamic code tracer. A dynamic code tracer for a program $\mathcal{P}$ will print out the instruction being executed, **as well as the state of the registers**, at every iteration. Although printr prints out the state of all 16 registers, **we will only be checking if r0 to r7, and r15 match the expected values.** A dynamic code trace is quite different than a static code trace because in a dynamic trace, especially in the presence of branches and jumps, because we actually follow the flow of instructions.

Given a program $\mathcal{P}$ written in *AWE*, you must write assembly code in *AWE* such that when your code is run, it will print out the dynamic trace of $\mathcal{P}$.

**Input**
You are provided a program $\mathcal{P}$ written in *AWE* that's 25 lines long. Like in Part 1, the only halt instruction is the 25th instruction. Your task is to write some assembly code $\mathcal{C}$ up to 100 lines long between addresses 0 to 99 (inclusive) that will print the dynamic code trace of $\mathcal{P}$.

**When we refer to dynamic code trace of $\mathcal{P}$, we're referring to the dynamic code trace if the initial pc value is 100, and the instructions of $\mathcal{P}$ are located between 100 and 124!!!!!** This is because like in Part 1, when we run execute your code, we will place your code $\mathcal{C}$ between lines 0 to 99, and then append on $\mathcal{P}$ starting from line 100. Refer to Part 1 again for a refresher on how we test/execute your code. For this part, **the program $\mathcal{P}$ will not contain any write, copy, print, or printr instructions, and it is guaranteed to terminate.** In addition, **$\mathcal{P}$ only uses registers r0 - r7, and r15.**

As a consequence of this, all branch and jump destinations of $\mathcal{P}$ are between 100 and 124, so that during execution, $\mathcal{P}$ will only jump to addresses within itself. In other words, imagine a program $\overline{\mathcal{P}}$ that contains $\mathcal{P}$ from lines 100 to 124, and is unspecified at all other memory addresses. If pc is 100 at the start of execution, then pc will never leave the range of $[100, 124]$. You may assume that when run in the manner described above, the program will always terminate.



PRINCETON Computer Science
Keller Center for Innovation in Engineering Education
CITADEL | CITADEL Securities
OLD MISSION
RADIX
SIG SUSQUEHANNA INTERNATIONAL GROUP, LLP
Evergreen Statistical Trading
PICSciE
ORFE Operations Research & Financial Engineering
Department of Mathematics
CENTER FOR STATISTICS AND MACHINE LEARNING
Electrical and Computer Engineering
MECHANICAL & AEROSPACE ENGINEERING
Princeton Physics




**Output**

Your program should print out two lines for each iteration it takes to execute $\mathcal{P}$ in the manner described above. The first line should be the instruction being executed, and the second should be the state of all the registers **after** the instruction has been executed. To print out this information, **use the printr instruction.** Do not print out the initial state of the registers before any instruction has been executed. You must print the final halt.

Consider the same example program $\mathcal{P}$ as in Part 1. A correct print output of your program $\mathcal{C}$ concatenated with $\mathcal{P}$ is

```
mov r0, 124
r0: 124, r1: 0, r2: 0, r3: 0, r4: 0, r5: 0, r6: 0, r7: 0, r8: 0, r9: 3,
   r10: 60, r11: 0, r12: 202, r13: 205, r14: 0, r15: 0
beq r2, r3, r0
r0: 124, r1: 0, r2: 0, r3: 0, r4: 0, r5: 0, r6: 0, r7: 0, r8: 1, r9: 1,
   r10: 60, r11: 0, r12: 208, r13: 320, r14: 124, r15: 0
halt
r0: 124, r1: 0, r2: 0, r3: 0, r4: 0, r5: 0, r6: 0, r7: 0, r8: 2, r9: 1,
   r10: 60, r11: 0, r12: 208, r13: 320, r14: 124, r15: 0
```

It's fine if your values for r8 to r14 are different than those in the sample output. We only require the instructions, and the values for r0 to r7, and r15 to match those in the sample output.



PRINCETON Computer Science
Keller Center for Innovation in Engineering Education
CITADEL | CITADEL Securities
OLD MISSION
RADIX
PACM
PRINCETON GEOSCIENCES
SIG SUSQUEHANNA INTERNATIONAL GROUP, LLP
Evergreen Statistical Trading
PICSciE
ORFE Operations Research & Financial Engineering
Department of Mathematics
CENTER FOR STATISTICS AND MACHINE LEARNING
Electrical and Computer Engineering
MECHANICAL & AEROSPACE ENGINEERING
Princeton Physics




## How to Submit

Email each part separately to coscon.submit@gmail.com. If you must resubmit, *respond to the thread where you sent your original submission; we cannot guarantee that your resubmission will be graded otherwise.*

**Part 1**
Store your assembly code (program $\mathcal{C}$ in the problem description) in a textfile named *Problem7aAssembly.txt*. Send an email with *exact* subject *Problem7aSubmission* and this file as an attachment.

**Part 2**
Store your assembly code (program $\mathcal{C}$ in the problem description) in a textfile named *Problem7bAssembly.txt*. Send an email with *exact* subject *Problem7bSubmission* and this file as an attachment.



PRINCETON Computer Science
Keller Center for Innovation in Engineering Education
CITADEL | CITADEL Securities
OLD MISSION
RADIX
PACM
PRINCETON GEOSCIENCES
SIG SUSQUEHANNA INTERNATIONAL GROUP, LLP
Evergreen Statistical Trading
PICSciE
ORFE Operations Research & Financial Engineering
Department of Mathematics
CENTER FOR STATISTICS AND MACHINE LEARNING
Electrical and Computer Engineering
MECHANICAL & AEROSPACE ENGINEERING
Princeton Physics