Hello, World

Things All Developers (and Other People) Should Know

Chapter Information

Learning Goals
1. Learn how to use GitHub
2. Learn how to install and use IntelliJ
3. Learn how to write an Hello World application in Java, run it from IntelliJ, from the command line and debug it
4. How to read from console using Scanner
5. How to do basic String operations
6. How to use variables and constants
7. Do Loop
8. Learn some of the basics of computer design and architecture
9. Comments and when to use them (and when NOT to use them)
10. Java Packages, versions

Final Project
Animation:
Show a single line animation on the console

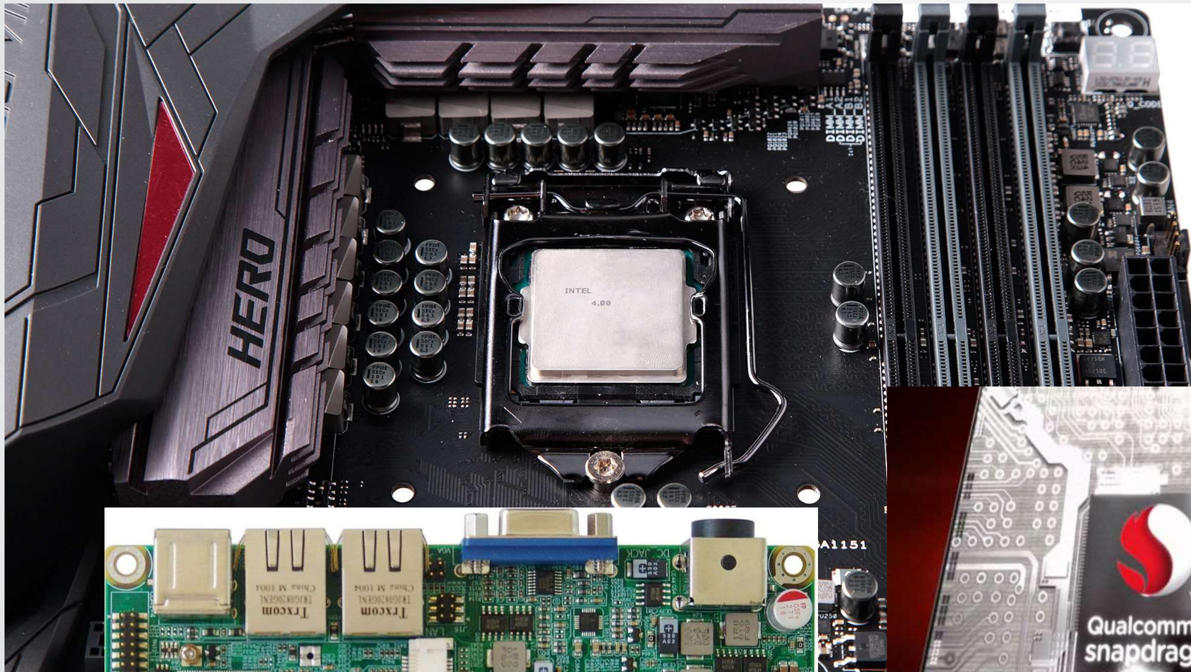
Tools to Install
1. Java SDK version 8 unless your team works with 11
2. IntelliJ (from JetBrains or local repository)
3. Create GitHub account
4. Git (from GitHub)
5. TortoiseGit (makes it easy to view repositories in Explorer)
6. Notepad++ (quick editor with syntax highlighting)

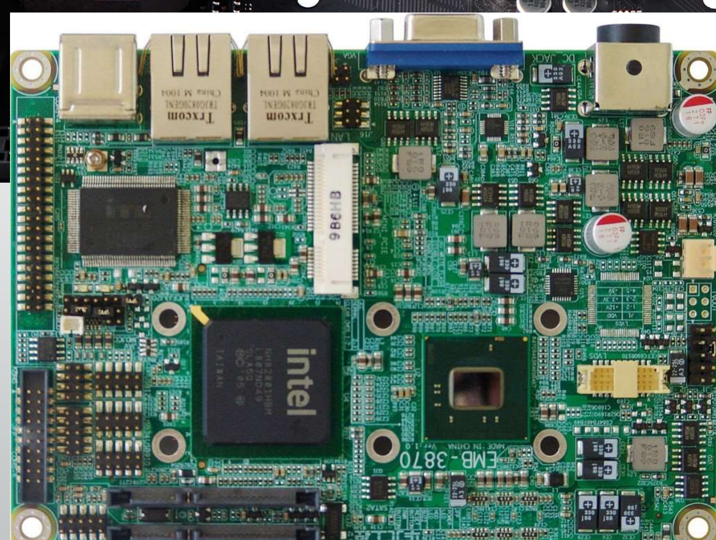
Computer Hardware

- **CPU**
- **Assembly Language**
- **Registers & Memory**
- **Virtual Memory**
- **Von Neumann Model**
- **I/O and Interrupts**
  - Disk I/O
  - Graphics
  - Networking

CPU

- Central Processing Unit
- Sits on Motherboard and controls most everything
- Processes ***Machine Instructions***
  - Recipes with predefined steps
    - Compare
    - Read and Write to memory
    - Jump around
    - Arithmetic


HERO
INTEL
4.00


intel
EMB-3870
Trxcom


Qualcomm
snapdragon

Assembly Language

- A very primitive programming language
  - Instructions that directly translate to machine language
  - Macros
- Very efficient
- Very hard to write long meaningful programs
- CPU specific
- Assembler: Assembly -> Machine Code

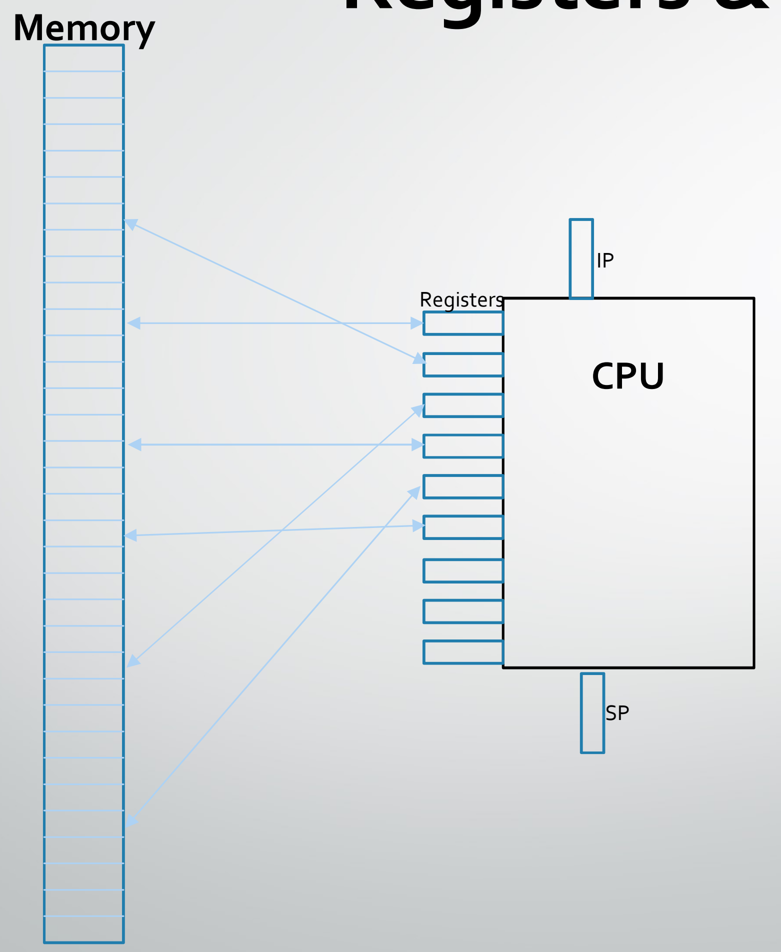
```
0x52ac76:  movl   7306562(%ebx), %eax
0x52ac7c:  movl   %eax, -20(%ebp)
0x52ac7f:  movl   $0, (%edi,%eax)
0x52ac86:  testl  %esi, %esi
0x52ac88:  je     0x52ad21                    ; -
    [UINavigationController _updateScrollViewFromViewController:
    toViewController:] + 425
0x52ac8e:  movl   7306542(%ebx), %eax
0x52ac94:  movl   (%edi,%eax), %eax
0x52ac97:  movl   %eax, -24(%ebp)
0x52ac9a:  movl   7212558(%ebx), %eax
0x52aca0:  movl   %eax, 4(%esp)
0x52aca4:  movl   %esi, (%esp)
0x52aca7:  calll  0x9bff06                    ; symbol stub for:
    objc_msgSend
0x52acac:  movl   %eax, -28(%ebp)
0x52acaf:  movl   %edx, -32(%ebp)        Thread 1: instruction step ov
0x52acb2:  movl   7211062(%ebx), %eax
0x52acb8:  movl   %eax, 4(%esp)
0x52acbc:  movl   %esi, (%esp)
```

```
MONITOR FOR 6802 1.4          9-14-80  TSC ASSEMBLER  PAGE    2

 C000                  ORG   ROM+$0000 BEGIN MONITOR
 C000 8E 00 70  START  LDS   #STACK

                ****************************************
                * FUNCTION: INITA - Initialize ACIA
                * INPUT: none
                * OUTPUT: none
                * CALLS: none
                * DESTROYS: acc A

 0013           RESETA EQU   %00010011
 0011           CTLREG EQU   %00000001

 C003 86 13     INITA  LDA A #RESETA    RESET ACIA
 C005 B7 80 04         STA A ACIA
 C008 86 11            LDA A #CTLREG    SET 8 BITS AND 2 STOP
 C00A B7 80 04         STA A ACIA

 C00D 7E C0 F1         JMP   SIGNON     GO TO START OF MONITOR

                ****************************************
                * FUNCTION: INCH - Input character
                * INPUT: none
                * OUTPUT: char in acc A
                * DESTROYS: acc A
                * CALLS: none
                * DESCRIPTION: Gets 1 character from terminal

 C010 B6 80 04  INCH   LDA A ACIA       GET STATUS
 C013 47               ASR A            SHIFT RDRF FLAG INTO CARRY
 C014 24 FA            BCC   INCH       RECIEVE NOT READY
 C016 B6 80 05         LDA A ACIA+1     GET CHAR
 C019 84 7F            AND A #$7F       MASK PARITY
 C01B 7E C0 79         JMP   OUTCH      ECHO & RTS

                ****************************************
                * FUNCTION: INHEX - INPUT HEX DIGIT
                * INPUT: none
                * OUTPUT: Digit in acc A
                * CALLS: INCH
                * DESTROYS: acc A
                * Returns to monitor if not HEX input

 C01E 8D F0     INHEX  BSR   INCH       GET A CHAR
 C020 81 30            CMP A #'0        ZERO
 C022 2B 11            BMI   HEXERR     NOT HEX
 C024 81 39            CMP A #'9        NINE
 C026 2F 0A            BLE   HEXRTS     GOOD HEX
 C028 81 41            CMP A #'A
 C02A 2B 09            BMI   HEXERR     NOT HEX
 C02C 81 46            CMP A #'F
 C02E 2E 05            BGT   HEXERR
 C030 80 07            SUB A #7         FIX A-F
 C032 84 0F     HEXRTS AND A #$0F       CONVERT ASCII TO DIGIT
 C034 39               RTS

 C035 7E C0 AF  HEXERR JMP   CTRL       RETURN TO CONTROL LOOP
```

Registers & Memory

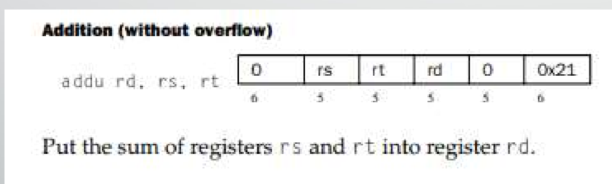
- Memory – holds binary data
- Manipulated by CPU
  - In Registers
  - Directly
- Registers
  - How CPU manipulates data
  - Special
- CPU Cache

Registers & Memory



Word!

- A **word** is the natural unit of data used by a particular processor design
- **A** comprises a set number bits N (8,9,16,32,64) as dictated by the processor design
- The word describes either
  - Data in the range $0..2^N\text{-}1$
  - Memory addresses in the range $0..2^N\text{-}1$
  - A representation of a machine code instruction


Addition (without overflow)

`addu rd, rs, rt`

0	rs	rt	rd	0	0x21
6	5	5	5	5	6

Put the sum of registers `rs` and `rt` into register `rd`.

Refresher on Radix Systems

Radix is a Latin word for "root". *Root* can be considered a synonym for *base* in the arithmetical sense.

In a system with radix *b* (*b* > 1), a string of digits $d_1 \ldots d_n$ denotes the number $d_1 b^{n-1} + d_2 b^{n-2} + \ldots + d_n b^0$, where $0 \leq d_i < b$.[1] In contrast to decimal, or radix 10, which has a ones' place, tens' place, hundreds' place, and so on, radix b would have a ones' place, then a b^1s' place, a b^2s' place, etc.[2]

2, 10, 16

Decimal {0..9} → $d_1\boldsymbol{10}^{n-1}+d_2\boldsymbol{10}^{n-2}+\ldots+d_n\boldsymbol{10}^0$

$77 = 77_{10}$

$= 7*10^1+7*10^0$

$= 70+7 = 77_{10}$

Binary {0..1} → $d_1\boldsymbol{2}^{n-1}+d_2\boldsymbol{2}^{n-2}+\ldots+d_n\boldsymbol{2}^0$

$0B01001101 = 01001101_2$

$= 0*2^7+1*2^6+0*2^5+0*2^4+1*2^3+1*2^2+0*2^1+1*1*2^0$

$= 2^6+2^3+2^2+1 = 77_{10}$

Hexadecimal {0..9,A..F} → $d_1\boldsymbol{16}^{n-1}+d_2\boldsymbol{16}^{n-2}+\ldots+d_n\boldsymbol{16}^0$

$0X4D = 4d_{16}$

$= 4*16^1+13*16^0$

$= 4*16+13 = 77_{10}$

Significance in Radix R

$$d_1 R^{n-1} + d_2 R^{n-2} + \ldots + d_n R^0 =$$

$$d_1 d_2 \ldots d_{n\ R}$$

Most Significant

Least Significant

A digit of a binary number in the context of computer systems is called a Bit (<u>Bi</u>nary Digi<u>t</u>)

We often talk about the most, and least significant bit, as indicated above…

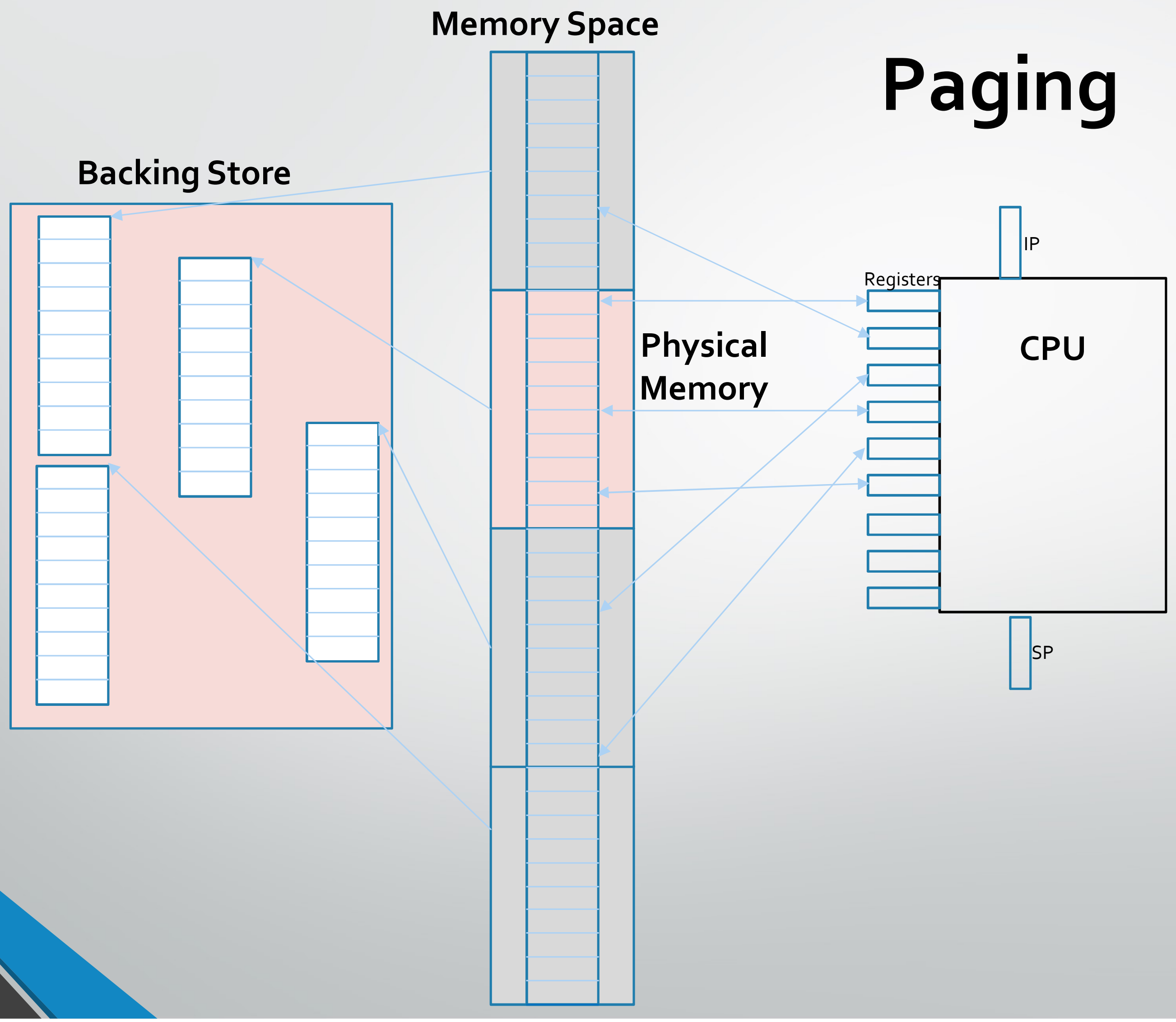
Virtual Memory

- Size of the address space
  - 2^{32} = 4,294,967,296

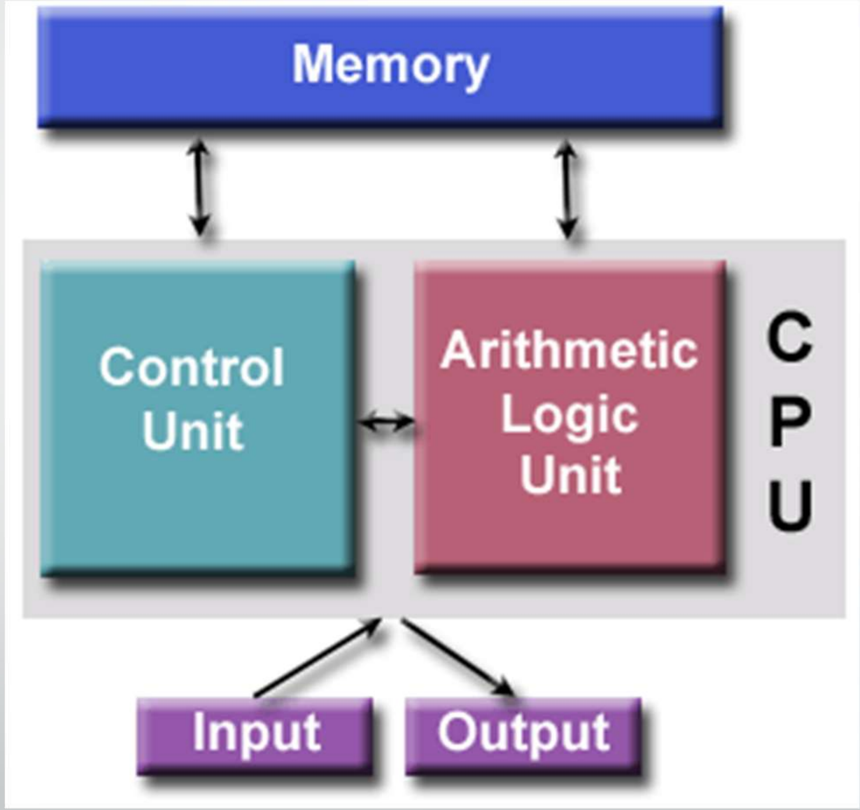
    (4 GigaGytes, 4 Billion Bytes)
  - 2^{64} = 18,446,744,073,709,551,616

    (18 ExaBytes, 18 Quintillion Bytes)
- Address spaces larger than 32 bit are impossible to maintain in physical memory.
  - OS and HW supports way less (2^{40})
  - Cost would be prohibitive (@1$/GB)
    - 18ExaBytes would cost $1,073,741,824
    - It would be huge, and draw a huge amount of power and cooling…
- Virtual Memory allows maintaining a large address space with much smaller physical memory

Paging


Memory Space
Backing Store
Physical Memory
Registers
IP
CPU
SP

Von Neumann Model



I/O and Interrupts

- Interrupts
- Memory mapped I/O
- Devices
  - Graphics
  - Disks
  - Network

Procedural Programming

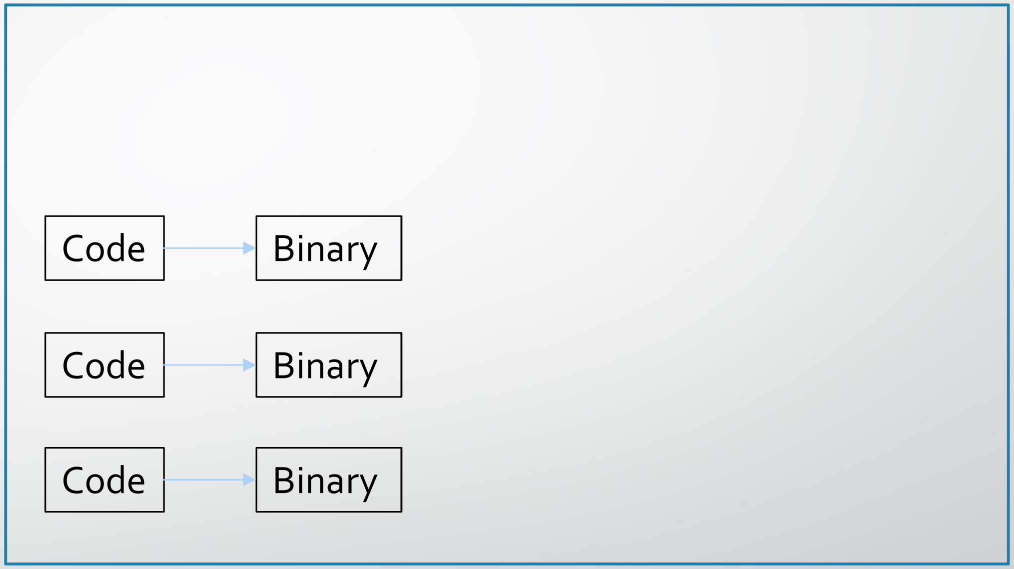
- **Some History**
- **What is Compiling**
  - Creating Binaries
- **What is Linking**
  - Putting things together
- **What is Loading**
  - Putting things where CPU can get them
- **Executing code**
  - Starting the program
- **Procedures and calling**
- **Basic Procedural Programming Constructs**

History

- 60's (Algol, ForTran, BASIC)
- 70's (Pascal, C)
- 80's (Ada)

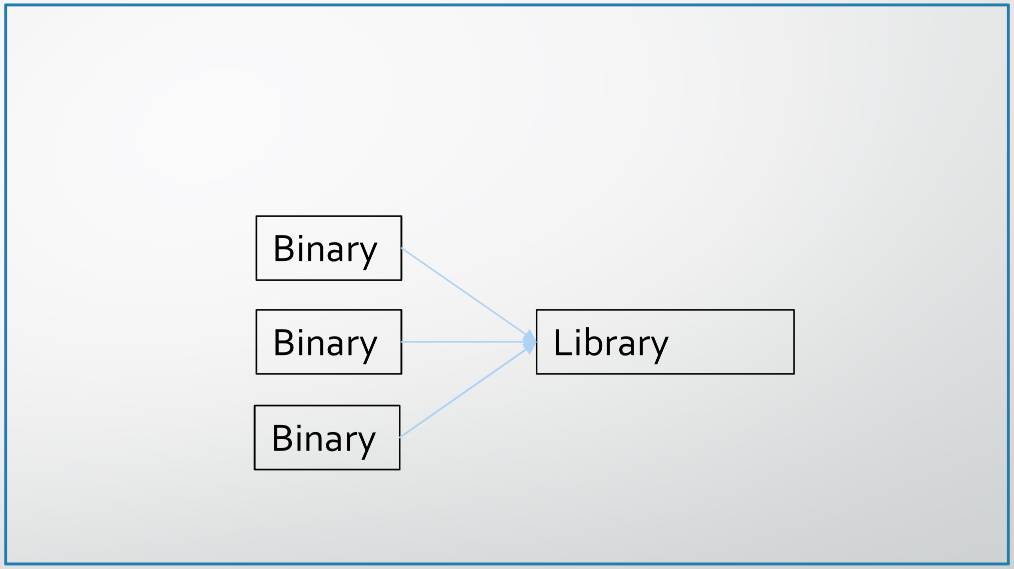
Compile

- Turn text (code) to binary



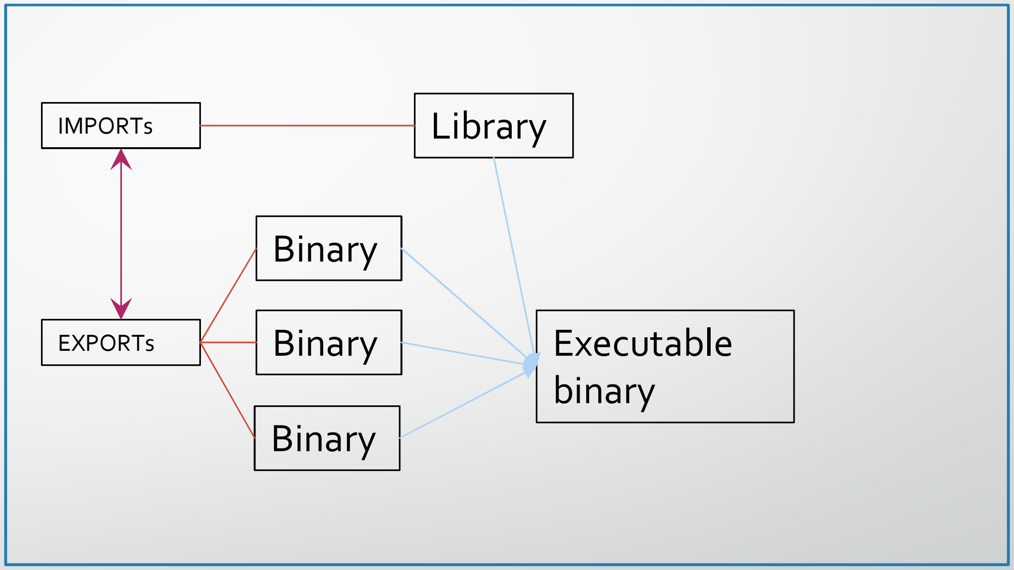
Lib

- Expose common abilities through a library



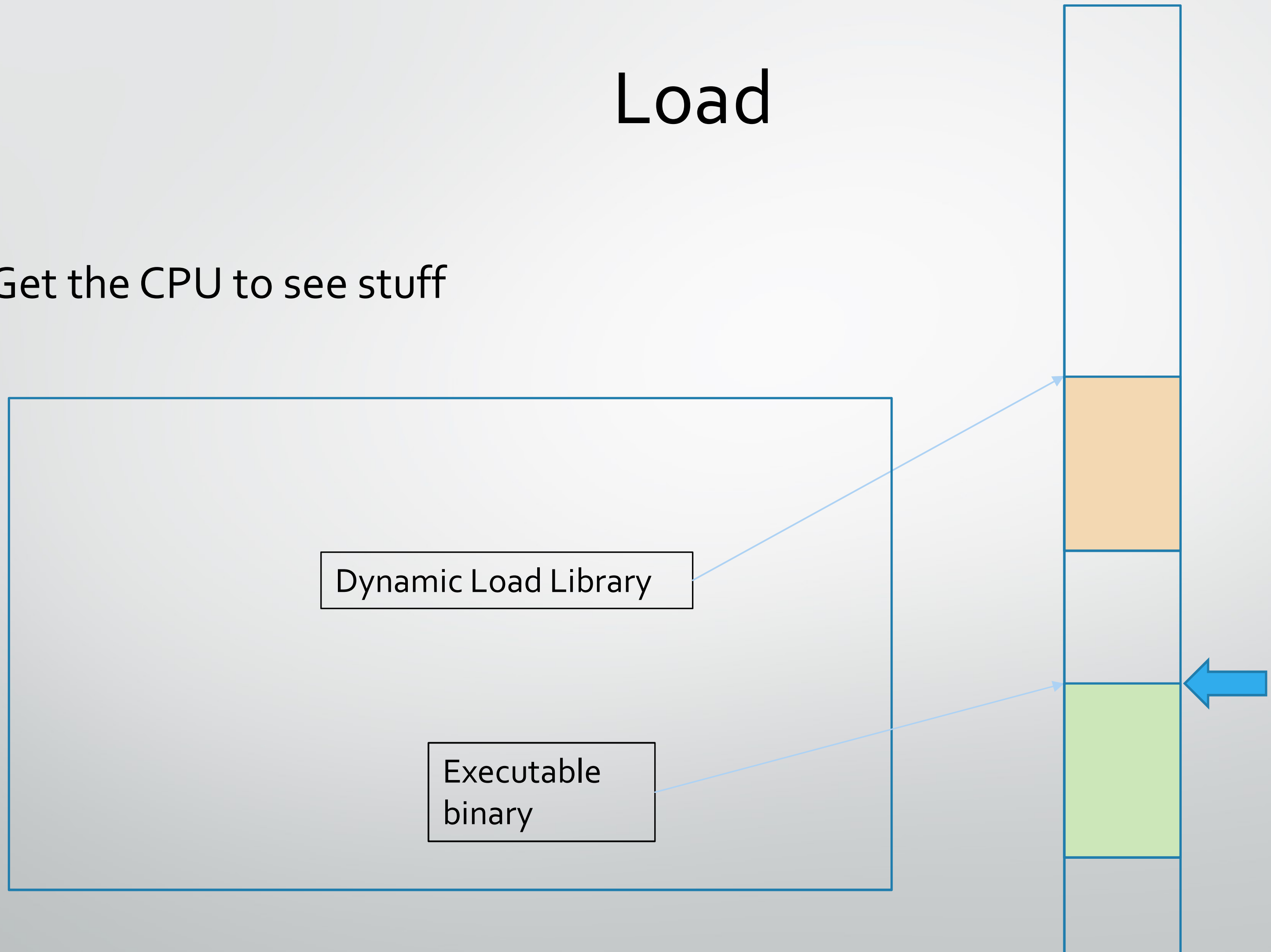
Link

- Bring things together
- Resolve IMPORTs through EXPORTs

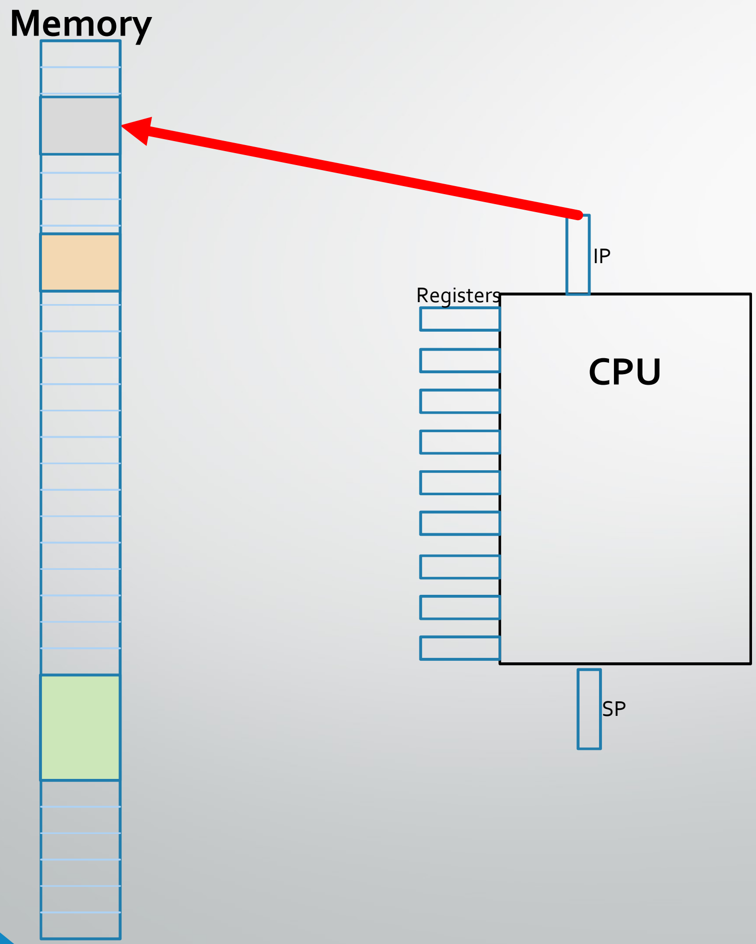


Load

- Get the CPU to see stuff



Running – The IP



Notation

- X = reference to X
    - Address
    - Register
- (X) = content of X

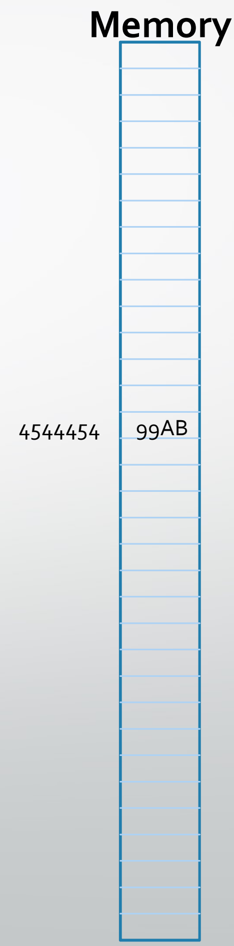
E.g.

R1

32

R1, 4544454

(R1), (4544454)



Instruction Processing by the CPU

Fetch the next instruction

CPU $\leftarrow$((IP))

IP = (IP) + 8

CPU Processes the fetched instruction

Example

R1 = 4000016

IP = 4000000

4000000	Mov R1,IP
4000008	Mov, 3, R1
4000016	Mov, 2, R1

Fetch the next instruction

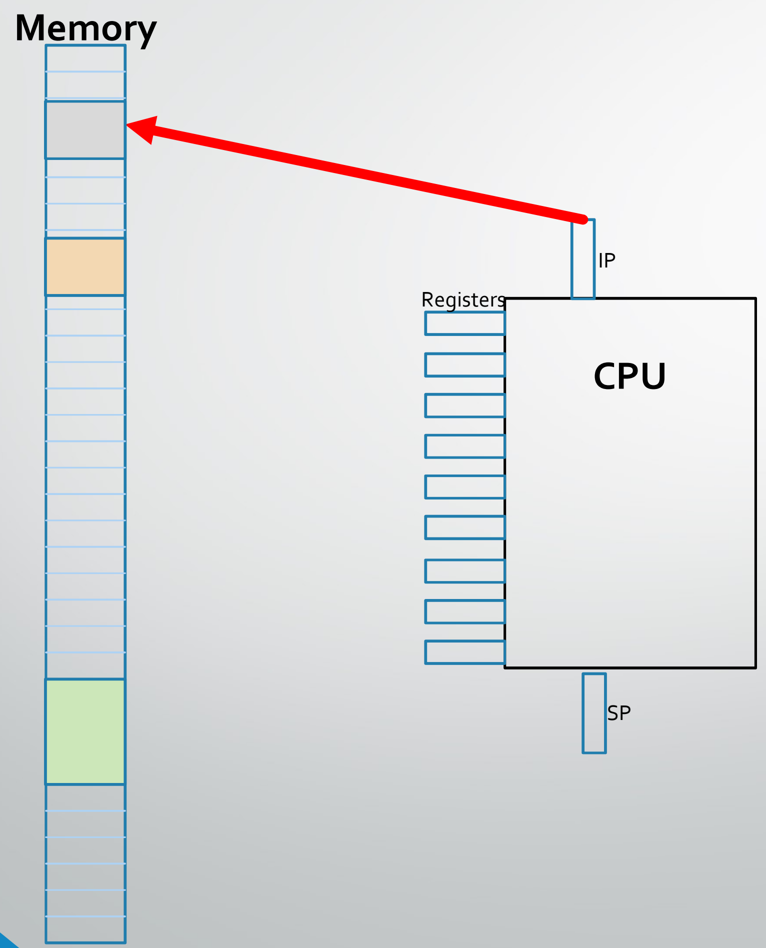
CPU ←((IP))

(IP) = (IP) + 8

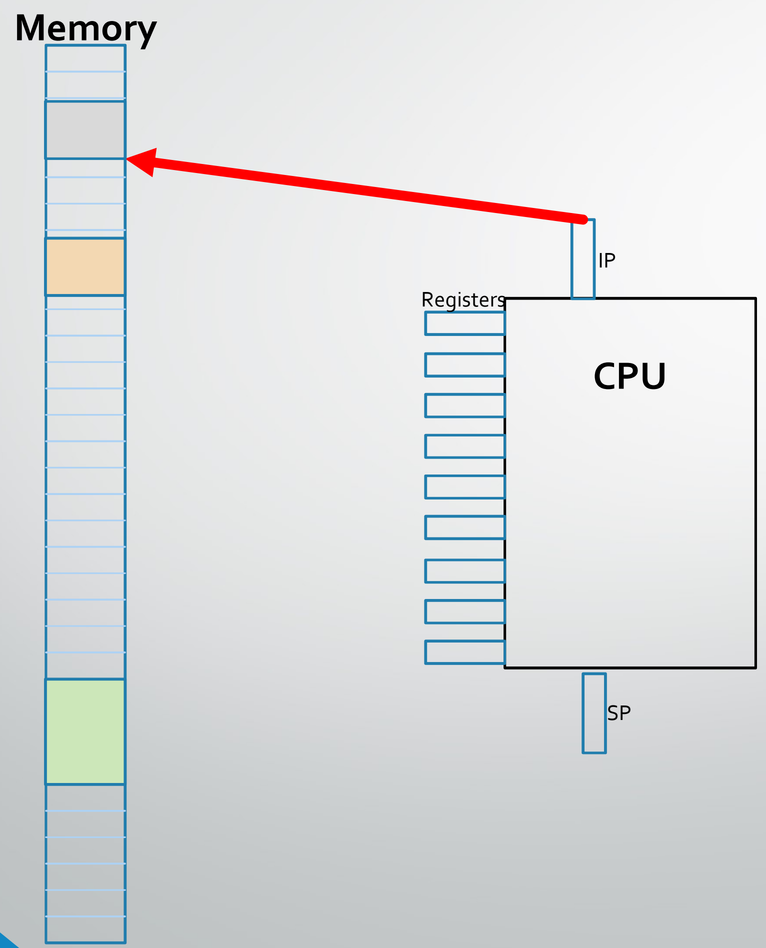
CPU processes the instruction

What are IP, R1 after FETCH?

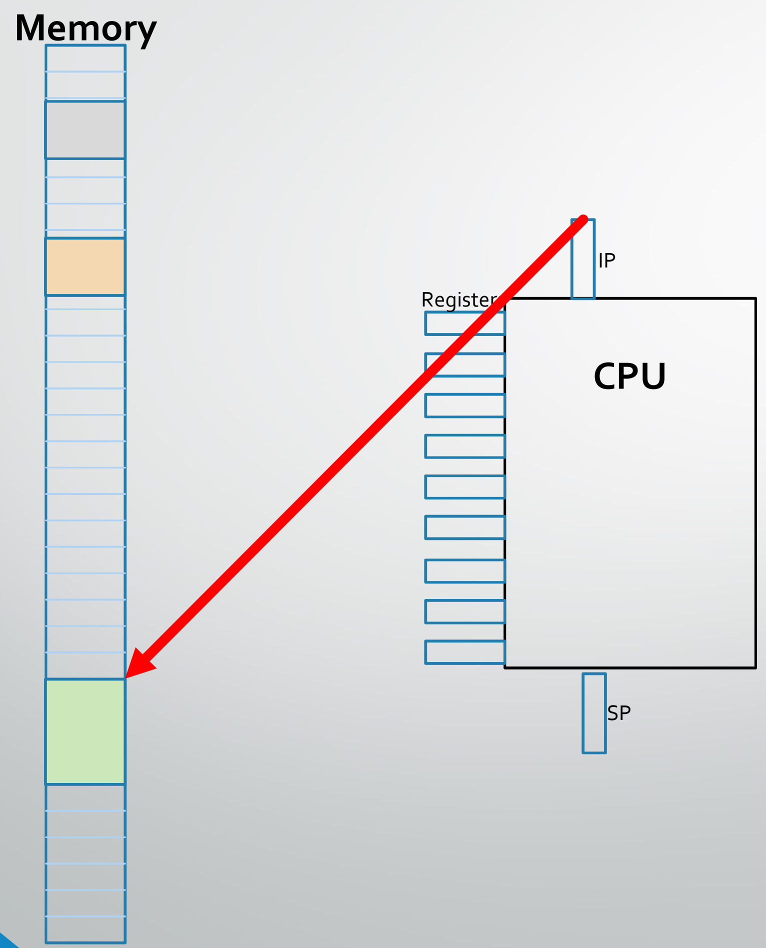
Running – The IP



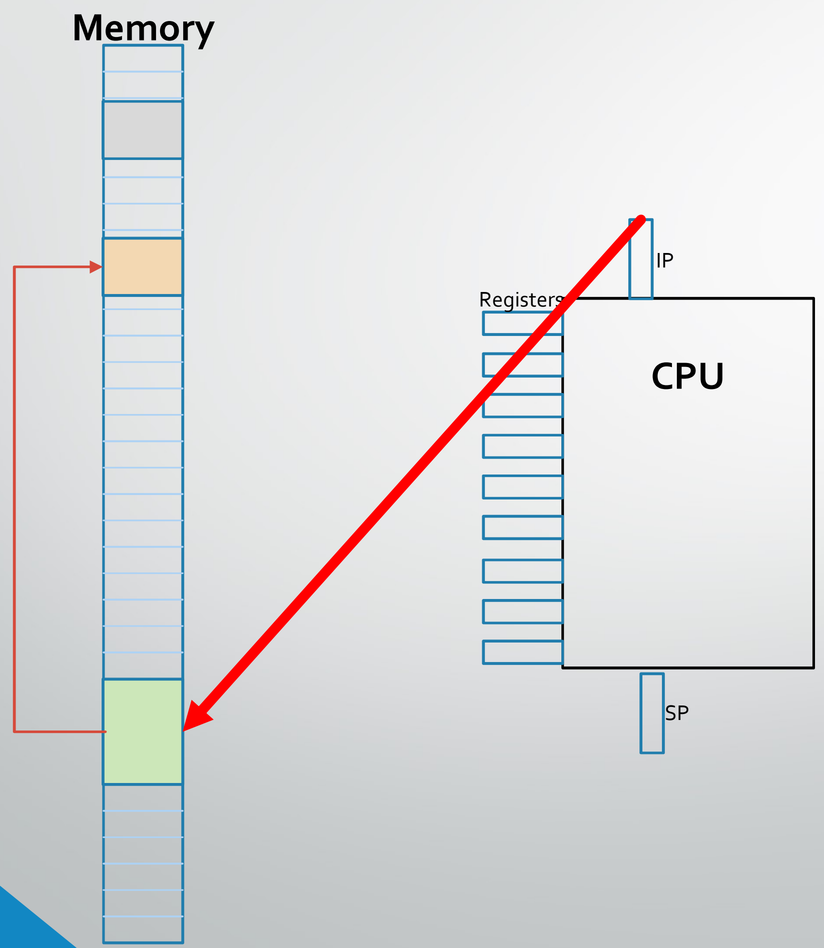
Running – The IP



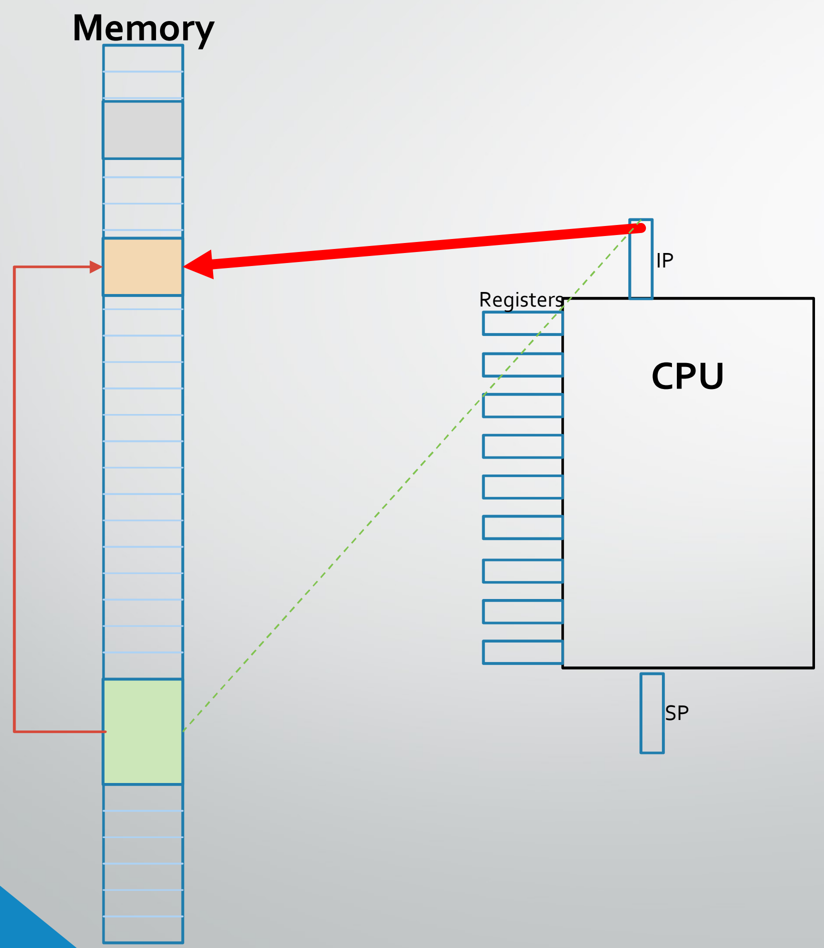
Running – The IP



Running – The IP



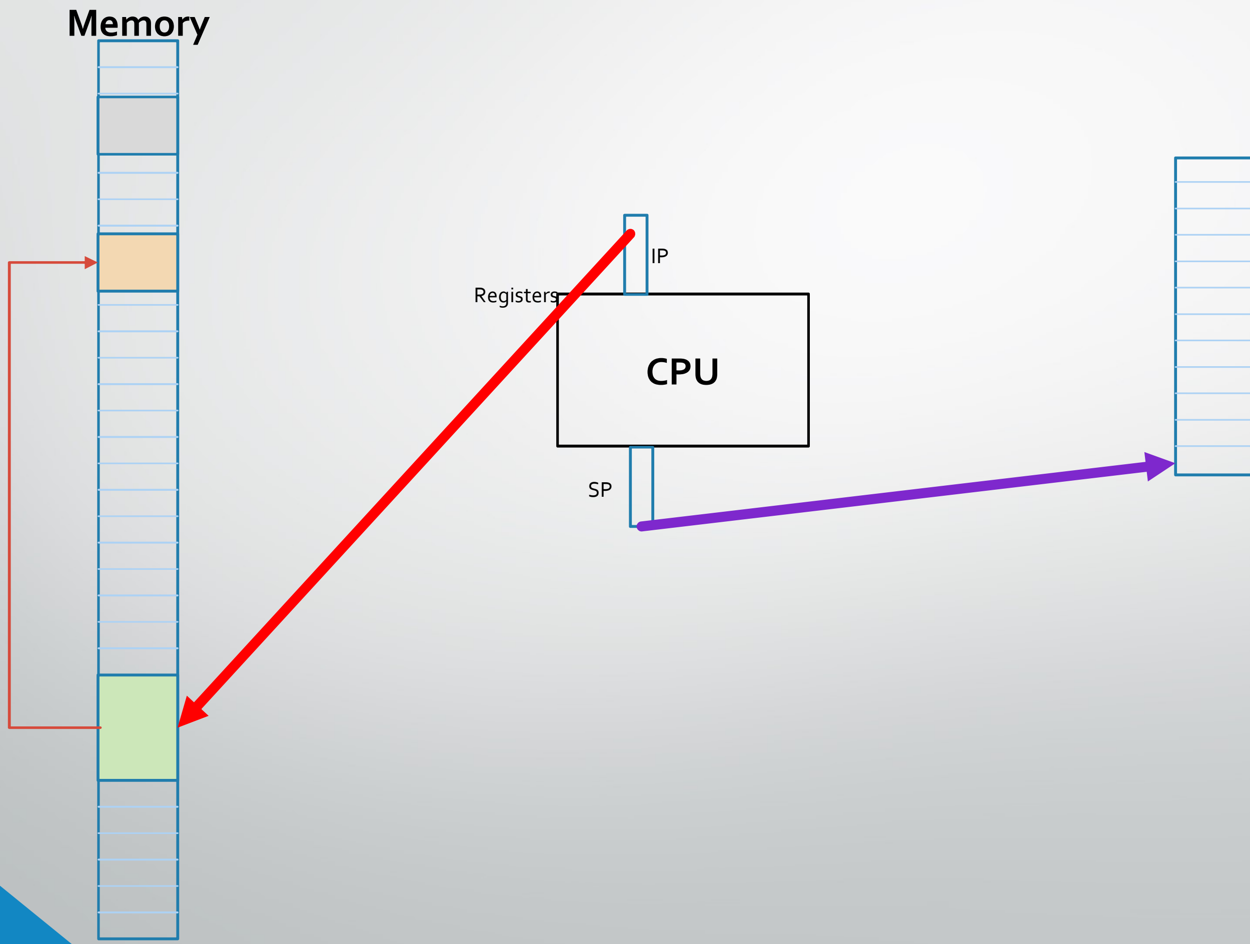
Calling a Procedure



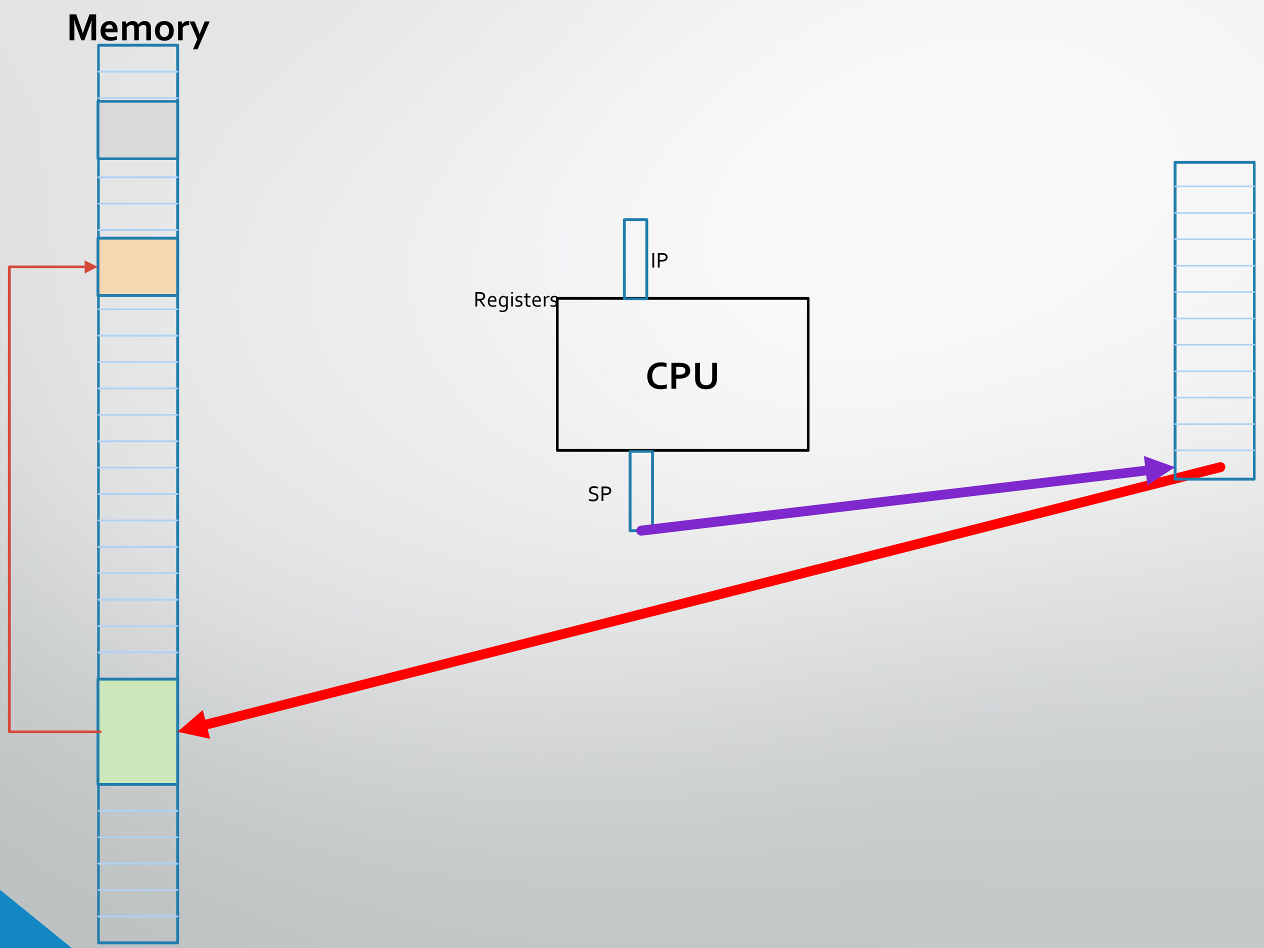
Procedures

- The Stack
- The Stack pointer

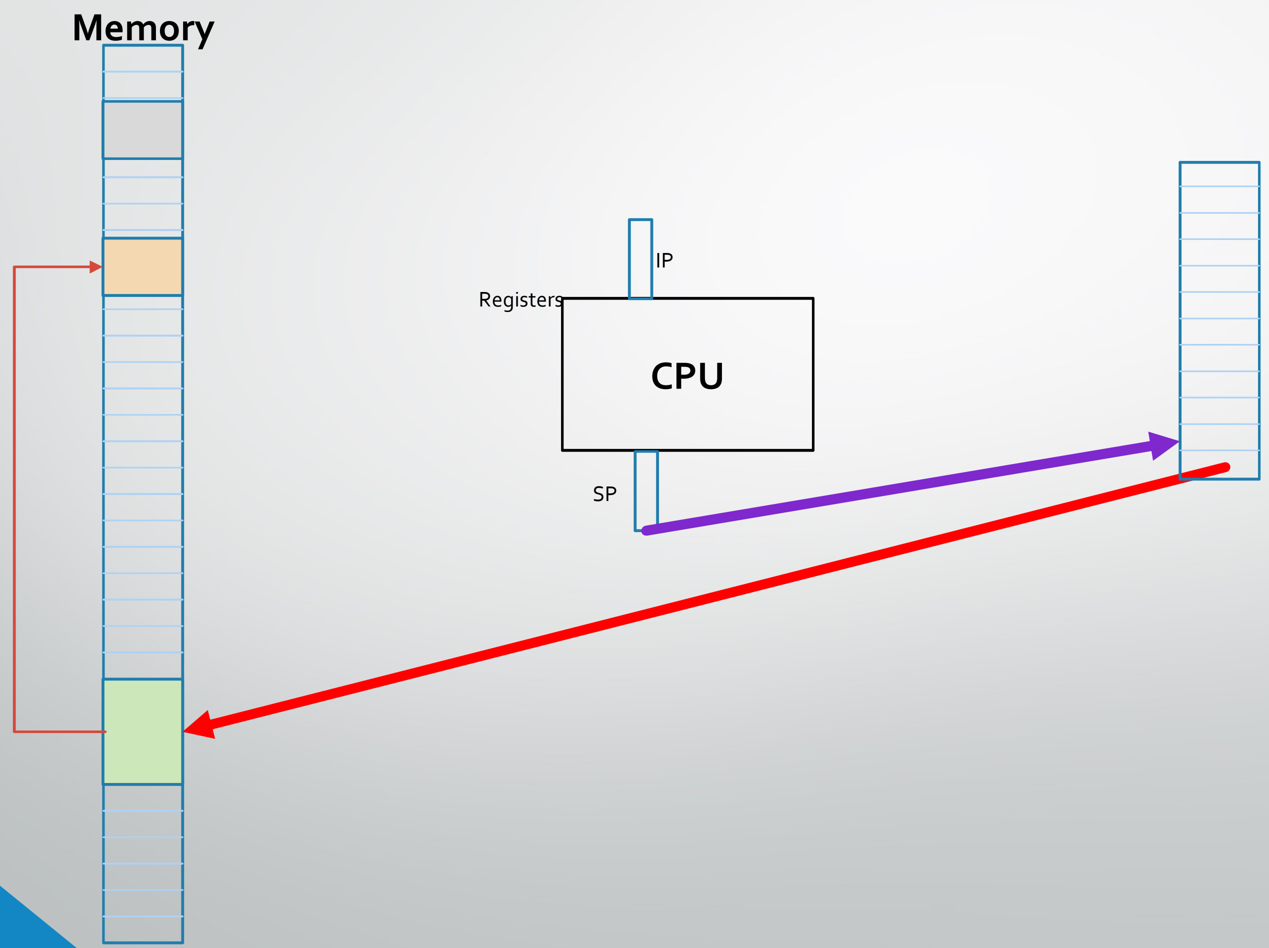
Calling a Procedure



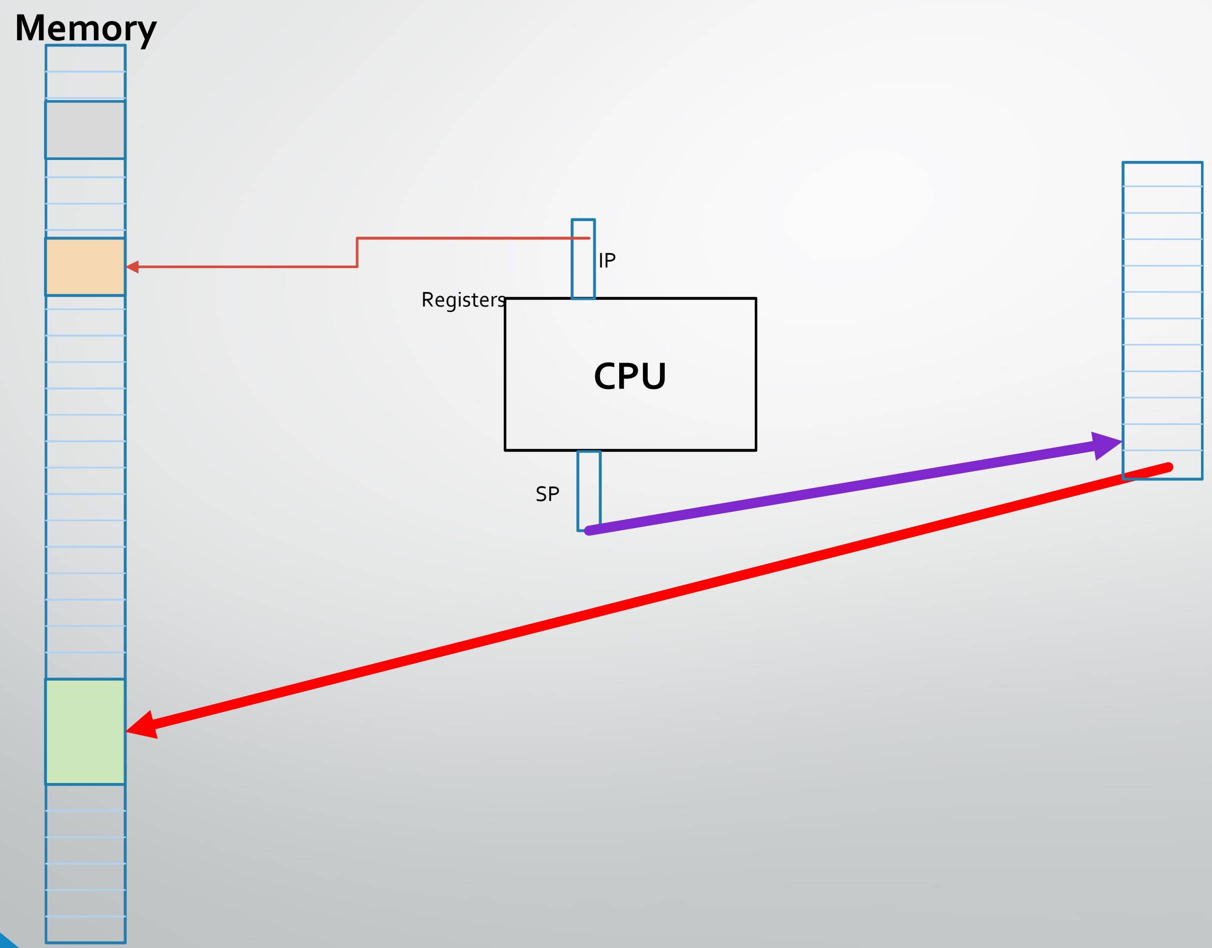
Calling a Procedure



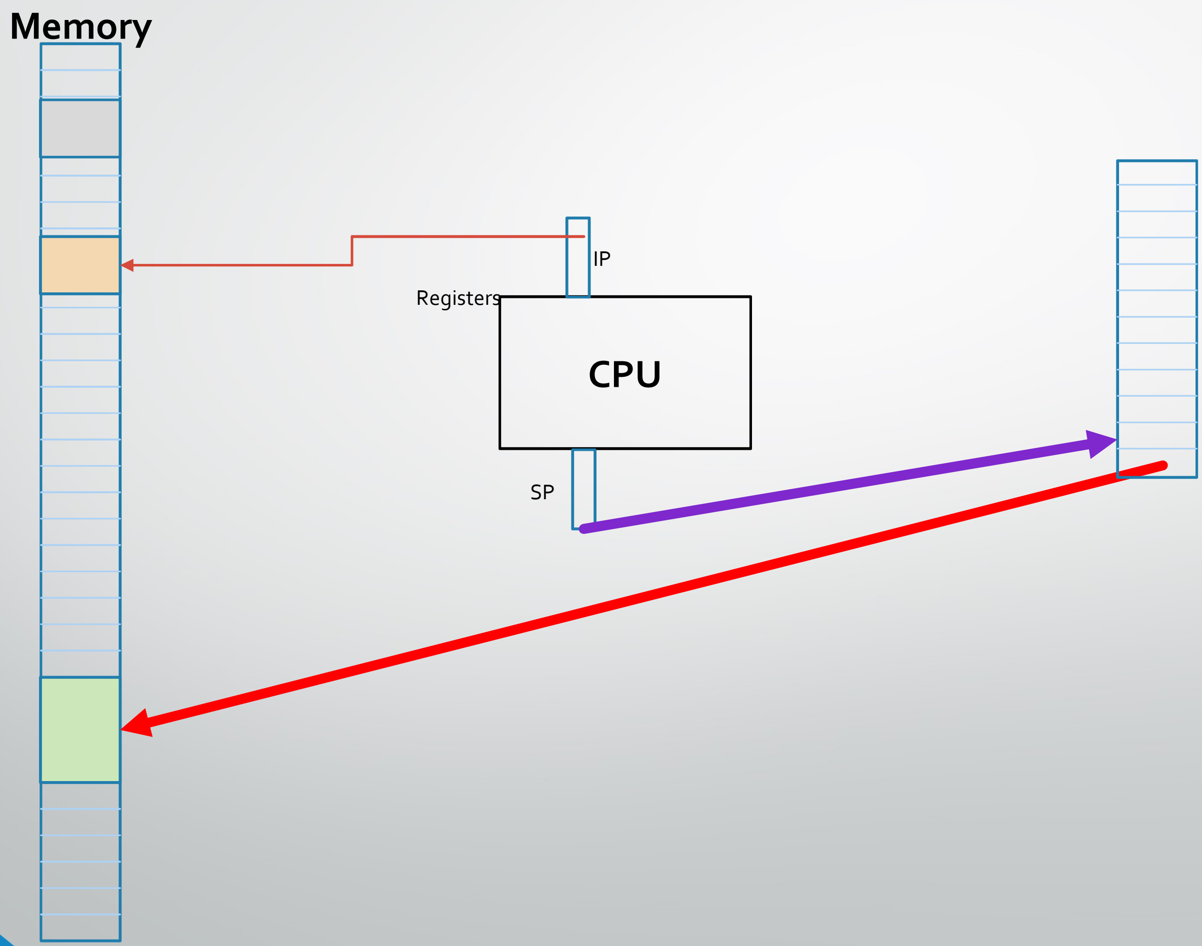
Calling a Procedure



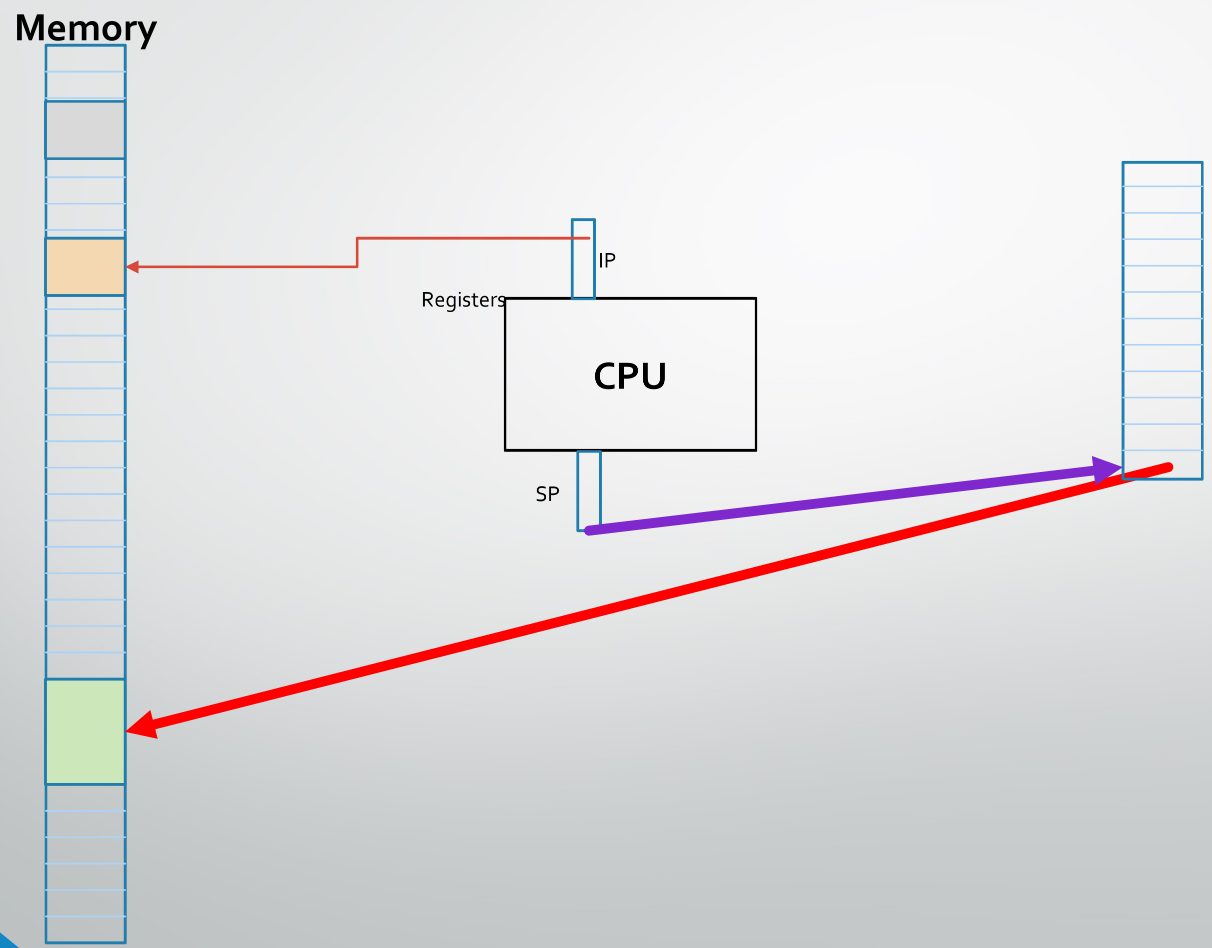
Calling a Procedure



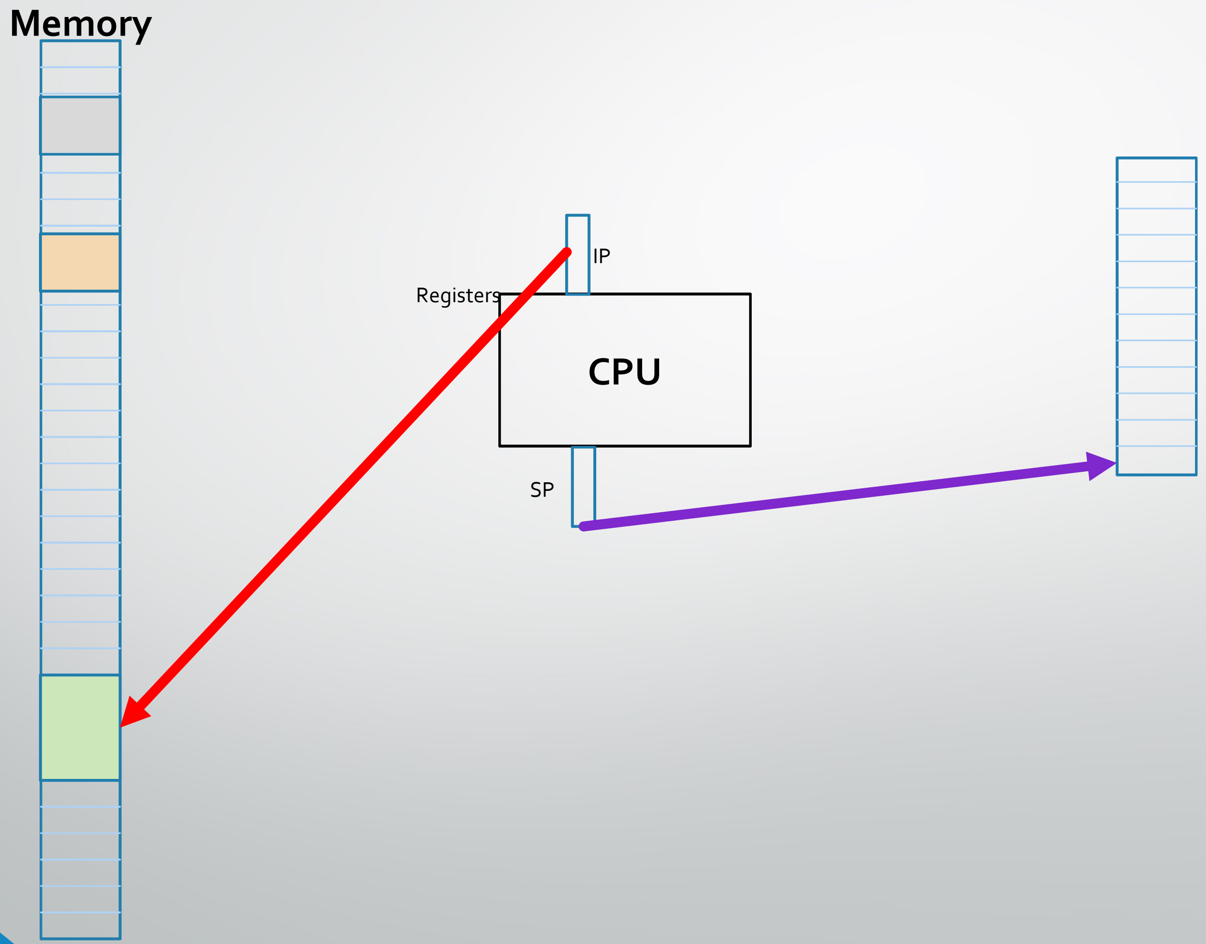
Return from Procedure



Return from Procedure



Return from Procedure



Logic / Arithmetic

- Usually executed by the CPU
- On data in registers
- Sometime on memory
- Depends on CPU

Assignment

- Move values from CPU registers into memory locations
- Sometime can move data from memory to memory
- Block transfers (BLOBs)
- Specific capabilities depend on CPU

ALGOL

```
BEGIN
FILE F (KIND=REMOTE);
EBCDIC ARRAY E [0:11];
REPLACE E BY "HELLO WORLD!";
WRITE (F, *, E);
END.
```

Fortran

```
program hello
print *, "Hello, World"
end program hello
```

BASIC

```
10 PRINT "Hello, World"

20 END
```

ADA

```
with Ada.Text_IO;

procedure HelloWorld is
   output_string : String(3..13);
begin
   output_string := "hello, world";
   Ada.Text_IO.Put (output_string);
end
```

C

```
#include <stdio.h>

int main (int argc, int argv)
{
   printf("hello, world\n");
   return 0;
}
```

Pascal

```
Program Hello(output)
Begin
   writeln("hello, world\n“);
End;
```

Conditionals

- Conditionals
  - Simple (if)
    - Special comparison operators in the CPU
    - Work on registers or on memory
  - Complex (switch)
    - Works with jumps: IP assignments

Memory Allocation

- Memory management subsystem
- Allocate()
- Free()
- Memory leaks if something goes wrong
- Most difficult part of “regular programming”
  - “Solved” through **managed memory systems** (later)