

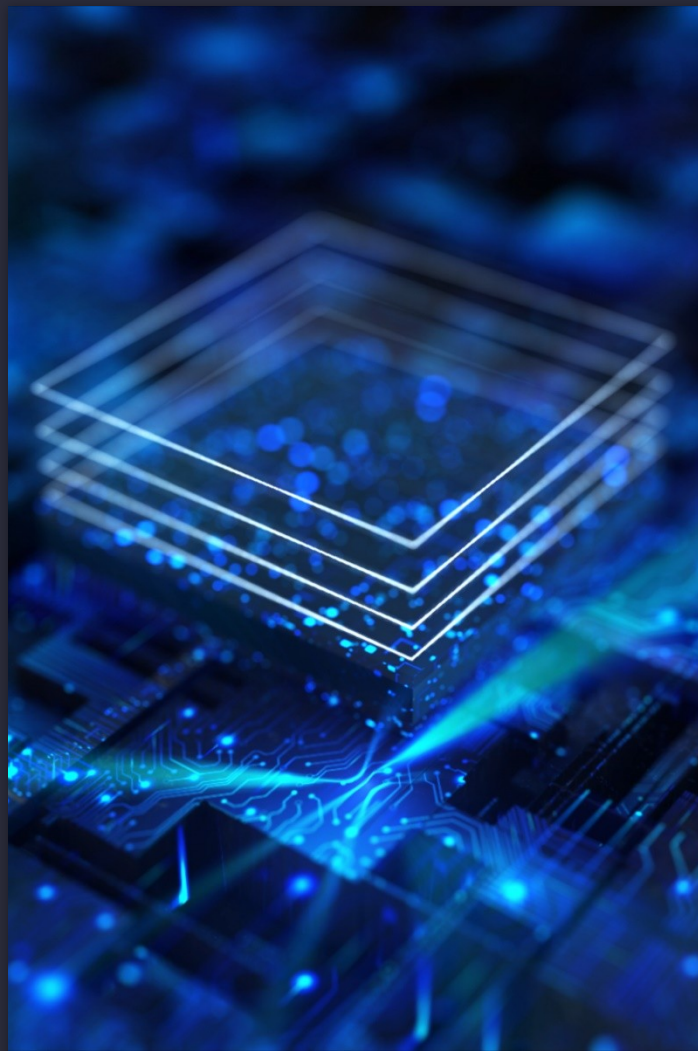
CDA3101



# Module 3: ARM Procedures (Part 1)



# Welcome!



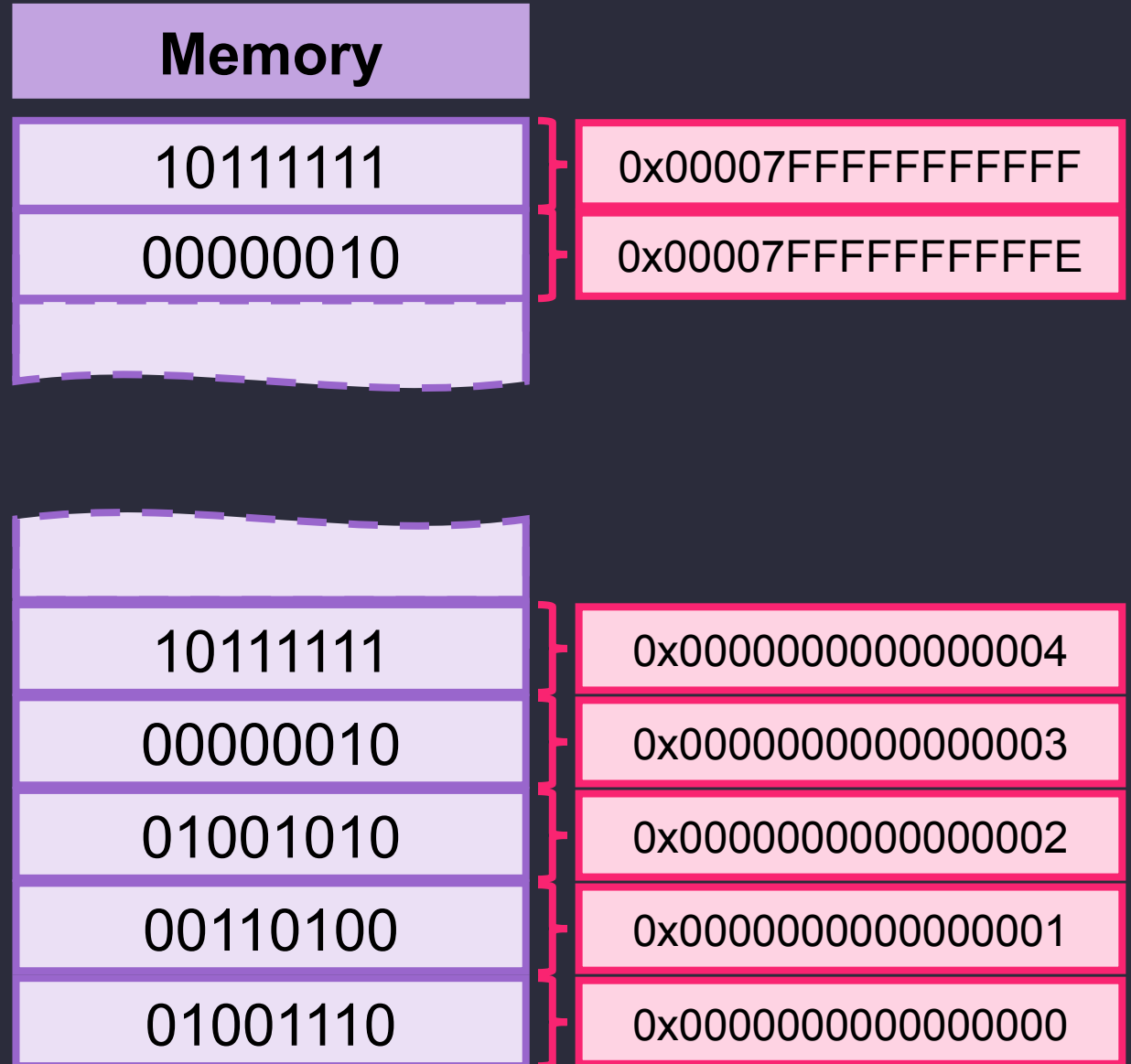


# ARM Memory Model

Memory is a linear array of  $2^{47}$  bytes.

Each byte has a 64-bit address and can store an 8-bit pattern.

ARM addresses are 64 bits long and range from 0 to  $2^{47}$ .





# ARM Memory Model

SP 0000 007f ffff fffC<sub>hex</sub>

**Stack**

**Dynamic Data**

**Static Data**

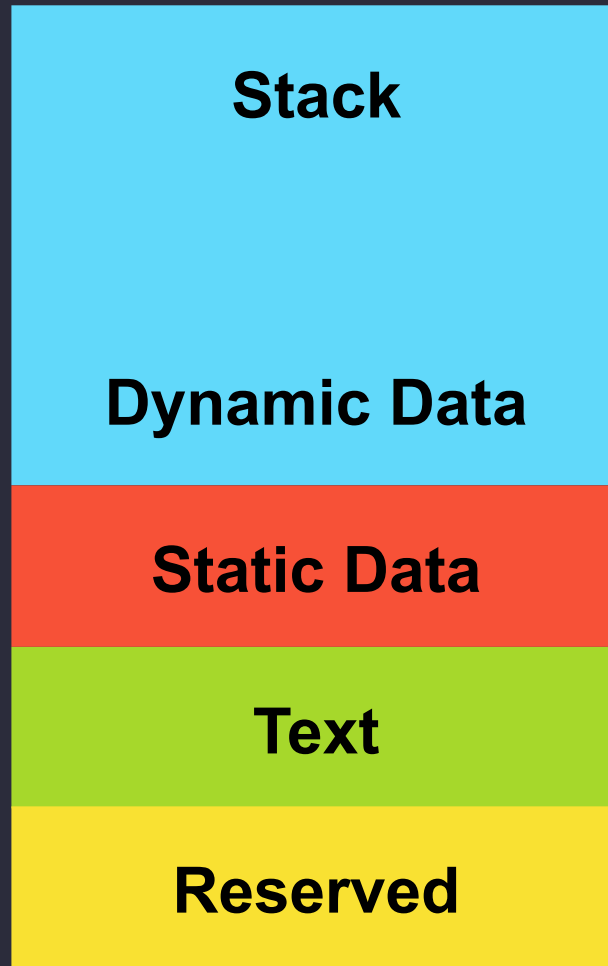
0000 0000 1000 0000<sub>hex</sub>

**Text**

PC 0000 0000 0040 0000<sub>hex</sub>

**Reserved**

0





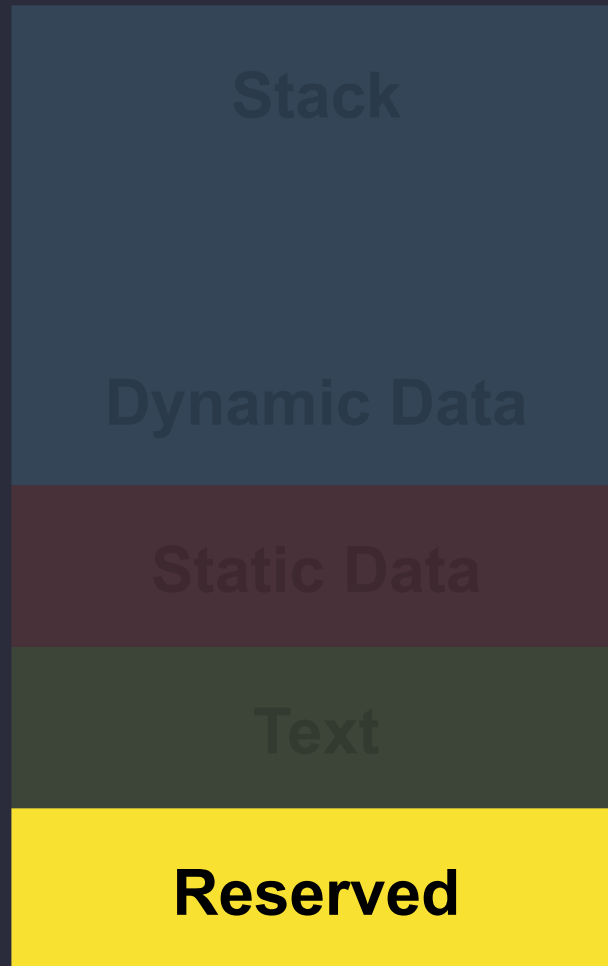
# ARM Memory Model

SP 0000 007f ffff fffC<sub>hex</sub>

0000 0000 1000 0000<sub>hex</sub>

PC 0000 0000 0040 0000<sub>hex</sub>

0



**Reserved**

Used by the operating system



# ARM Memory Model

SP 0000 007f ffff fffC<sub>hex</sub>

0000 0000 1000 0000<sub>hex</sub>

PC 0000 0000 0040 0000<sub>hex</sub>

0



Stores code for user programs



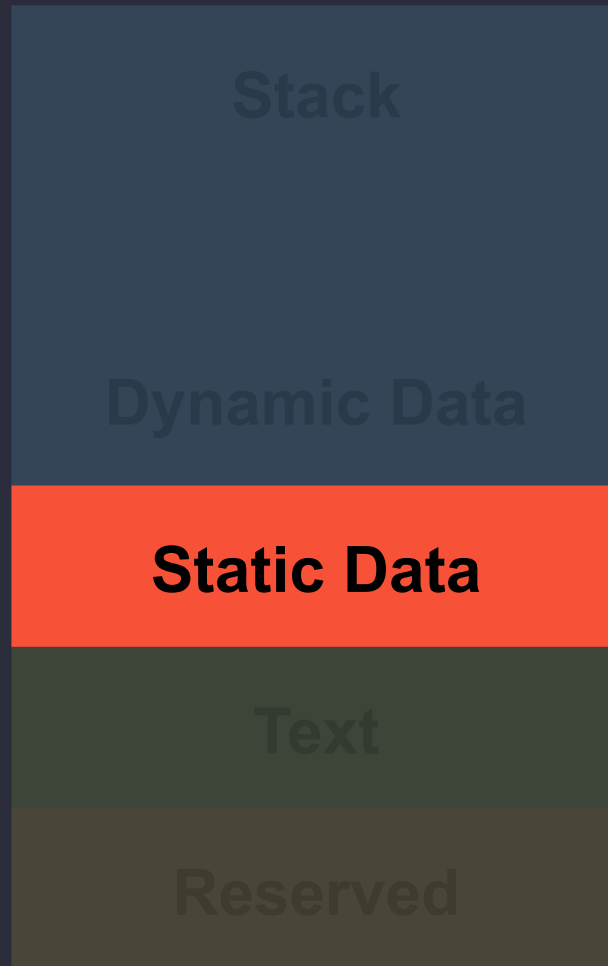
# ARM Memory Model

SP 0000 007f ffff fffC<sub>hex</sub>

0000 0000 1000 0000<sub>hex</sub>

PC 0000 0000 0040 0000<sub>hex</sub>

0



Stores data early-bound by compiler  
(static variables)



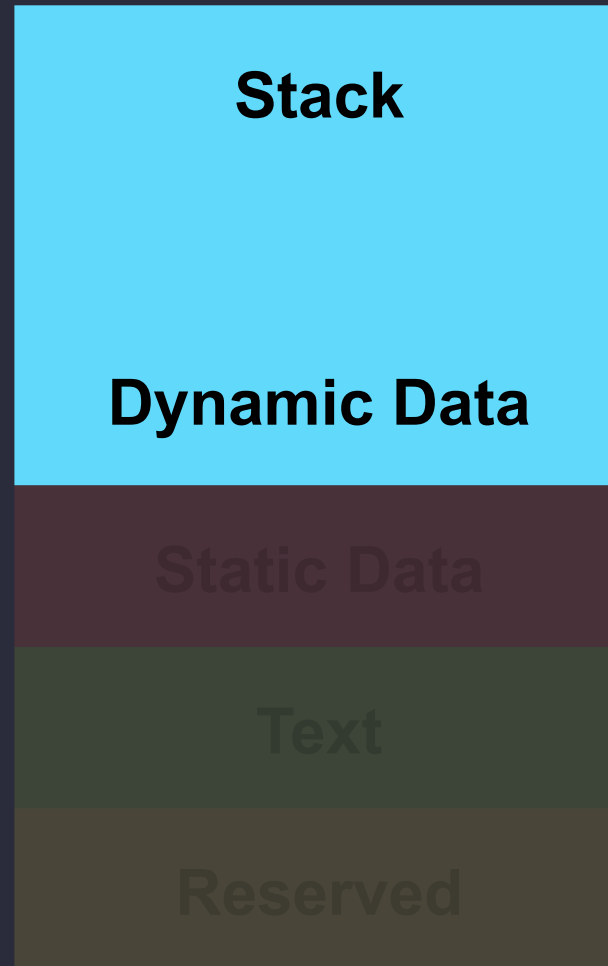
# ARM Memory Model

SP 0000 007f ffff fffc<sub>hex</sub>

0000 0000 1000 0000<sub>hex</sub>

PC 0000 0000 0040 0000<sub>hex</sub>

0



Stores dynamic  
program data structures  
(local variables, return addresses)

## Heap

Stores dynamic user data structures  
(memory allocated for  
reference-based data structures)





# Stored Program Concept

The **Text** segment stores code for user programs.

Each machine code instruction is 32 bits (4 bytes).

The instruction's address will be the address of the first byte.

Subsequent instructions will have an address that is different by 4.

**Processor**

## Memory

**Accounting Program**  
(machine code)

**Editor Program**  
(machine code)

**C Compiler**  
(machine code)

0x0000000010000000

0x0000000000400000



# Stored Program Concept

Instructions are **word aligned**.  
(word = 4 bytes)

Instructions must have an  
address divisible by 4.

**Program Counter (PC)** =  
address of the instruction  
being executed

Add 4 if executing next instruction

Add offset if executing a branch

**Processor**

## Memory

**Accounting Program**  
(machine code)

**Editor Program**  
(machine code)

**C Compiler**  
(machine code)

0x0000000010000000

0x0000000000400000



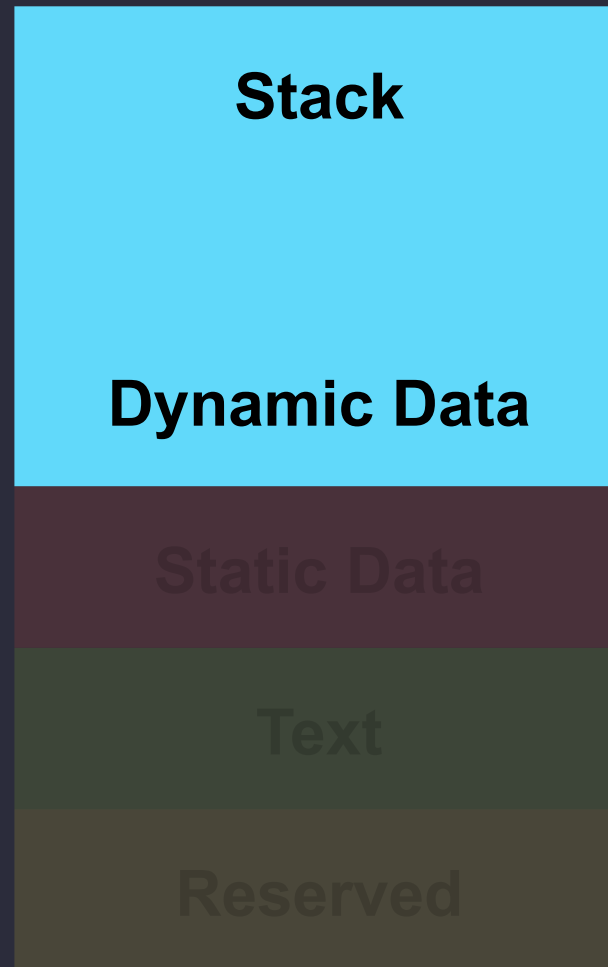
# Stack Frame

SP 0000 007f ffff fffc<sub>hex</sub>

0000 0000 1000 0000<sub>hex</sub>

PC 0000 0000 0040 0000<sub>hex</sub>

0



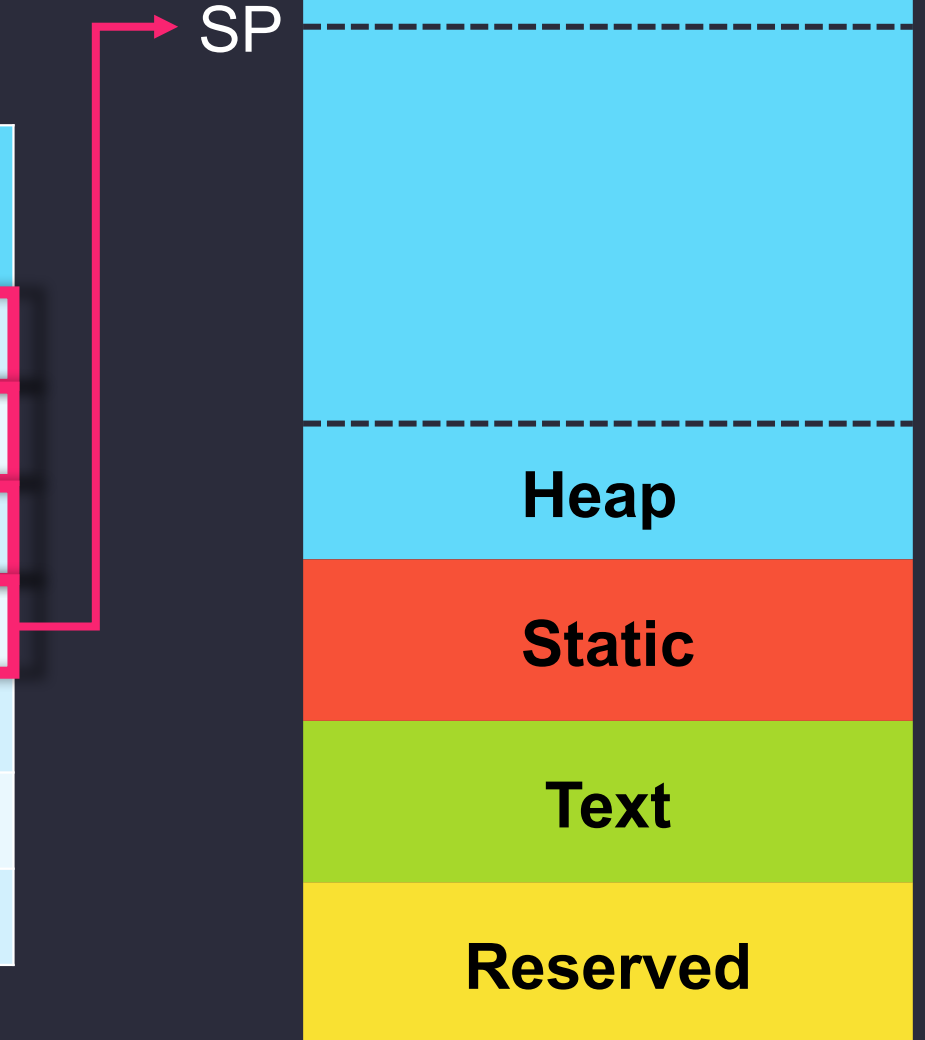
## Stack Frame

A block of data that contains register contents and variables' values that you want to save when the procedure is called (so the data is not lost)



# ARM Registers

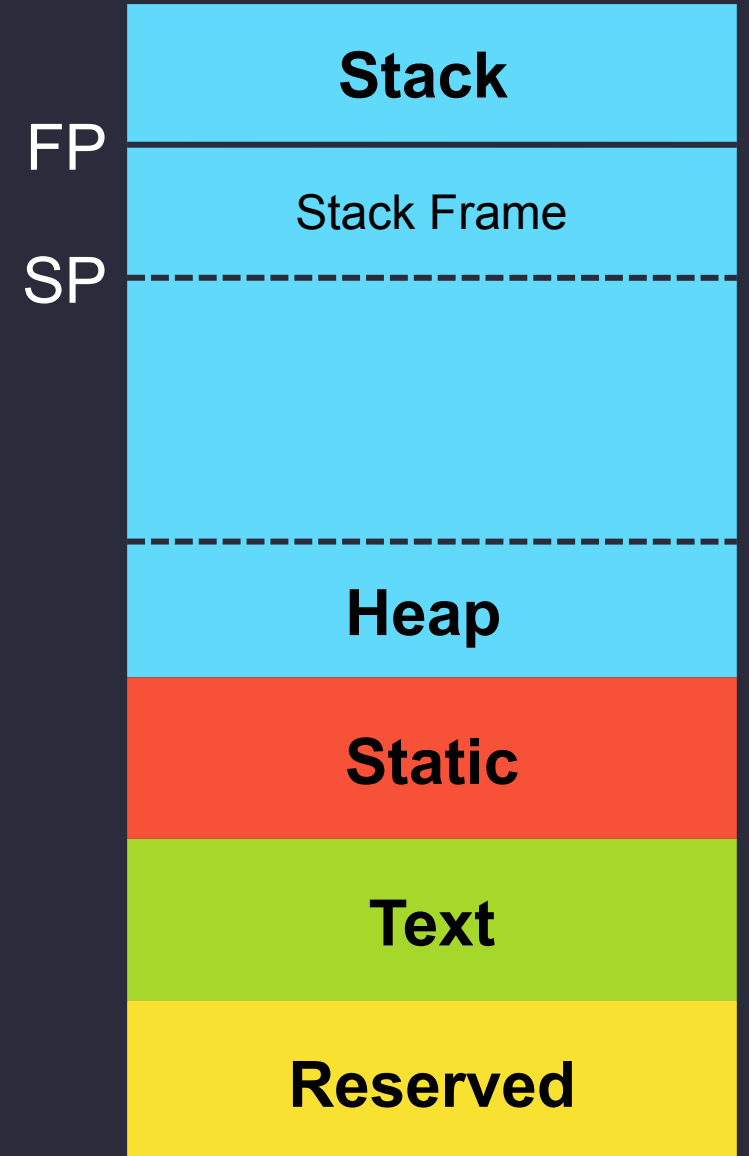
Name	#	Use	Preserved across function call?
X0-X7	0-7	Arguments/results	No
X9-X15	9-15	Temporaries	No
X19-X27	19-27	Saved	Yes
SP	<b>28</b>	Stack pointer	Yes
FP	29	Frame pointer	Yes
X30	30	Return address	Yes
XZR	31	The constant zero	Yes





# ARM Registers

Name	#	Use	Preserved across function call?
X0-X7	0-7	Arguments/results	No
X9-X15	9-15	Temporaries	No
X19-X27	19-27	Saved	Yes
SP	<b>28</b>	Stack pointer	Yes
FP	29	Frame pointer	Yes
X30	30	Return address	Yes
XZR	31	The constant zero	Yes





# Translating Procedure Calls

```
int procedureA(int myArgument)
{
    int returnVariable, anotherVariable=2;

    returnVariable =
    procedureB(anotherVariable);
    myArgument += returnVariable;
    return myArgument;
}
```

0x0000000010000000

0x0000000001000004

```
int procedureB(int argument)
{
    int a;
    //code here
    return a;
}
```

When calling myProcedure, the return address 0x00000000000100004 is put in register x30.

When this line executes, it returns to the address found in register x30.



# Translating Procedure Calls

Arguments

```
int procedureA(int myArgument, int
anotherArgument)
{
    int returnVariable, anotherVariable=2;

    returnVariable =
    procedureB(anotherVariable);
    myArgument += returnVar
    return myArgument;
}

int procedureB(int argument)
{
    int a;
    //code here
    return a;
}
```

Result

Argument

Result



# Translating Procedure Calls

**Arguments**

These values will be in registers X0 and X1.

```
int procedureA(int myArgument, int
anotherArgument)
{
    int returnVariable, anotherVariable=2;

    returnVariable =
    procedureB(anotherVariable);
    myArgument += returnVar
    return myArgument;
}
```

**Result**

This value will be placed in register X2.

**Argument**

This value will be in register X0.

```
int procedureB(int argument)
{
    int a;
    //code here
    return a;
}
```

**Result**

This value will be placed in register X1.



# Translating Procedure Calls

```
int procedureA(int myArgument, int  
anotherArgument)  
{
```

Caller

```
    int returnVariable, anotherVariable=2;
```

```
    returnVariable =  
    procedureB(anotherVariable);  
    myArgument += returnVariable;  
    return myArgument;  
}
```

Callee

```
int procedureB(int argument)  
{  
    int a;  
    //code here  
    return a;  
}
```

Stack

SP

# Translating Procedure Calls

```
int procedureA(int myArgument, int  
anotherArgument)  
{
```

Caller

```
    int returnVariable, anotherVariable=2;
```

```
    returnVariable =  
    procedureB(anotherVariable);  
    myArgument += returnVariable;  
    return myArgument;  
}
```

Callee

```
int procedureB(int argument)  
{  
    int a;  
    //code here  
    return a;  
}
```

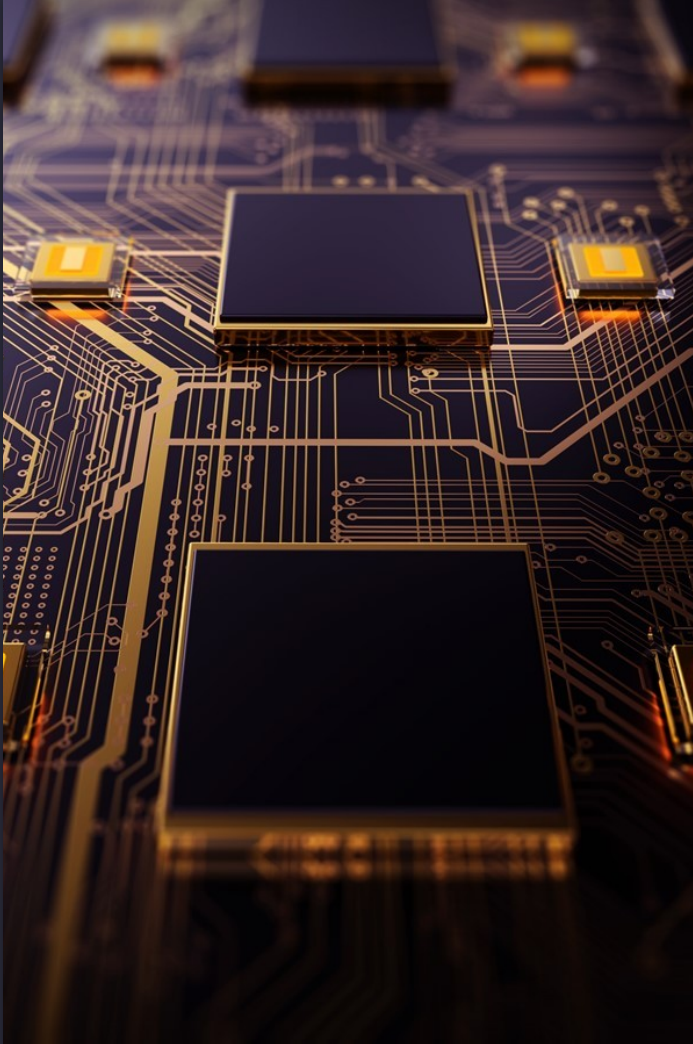
Stack

Current return address (X30)  
myArgument (X0),  
anotherArgument (X1),  
anotherVariable(X9)

SP



# Wrap Up



Thank you for watching.



**UF** | UNIVERSITY of  
FLORIDA