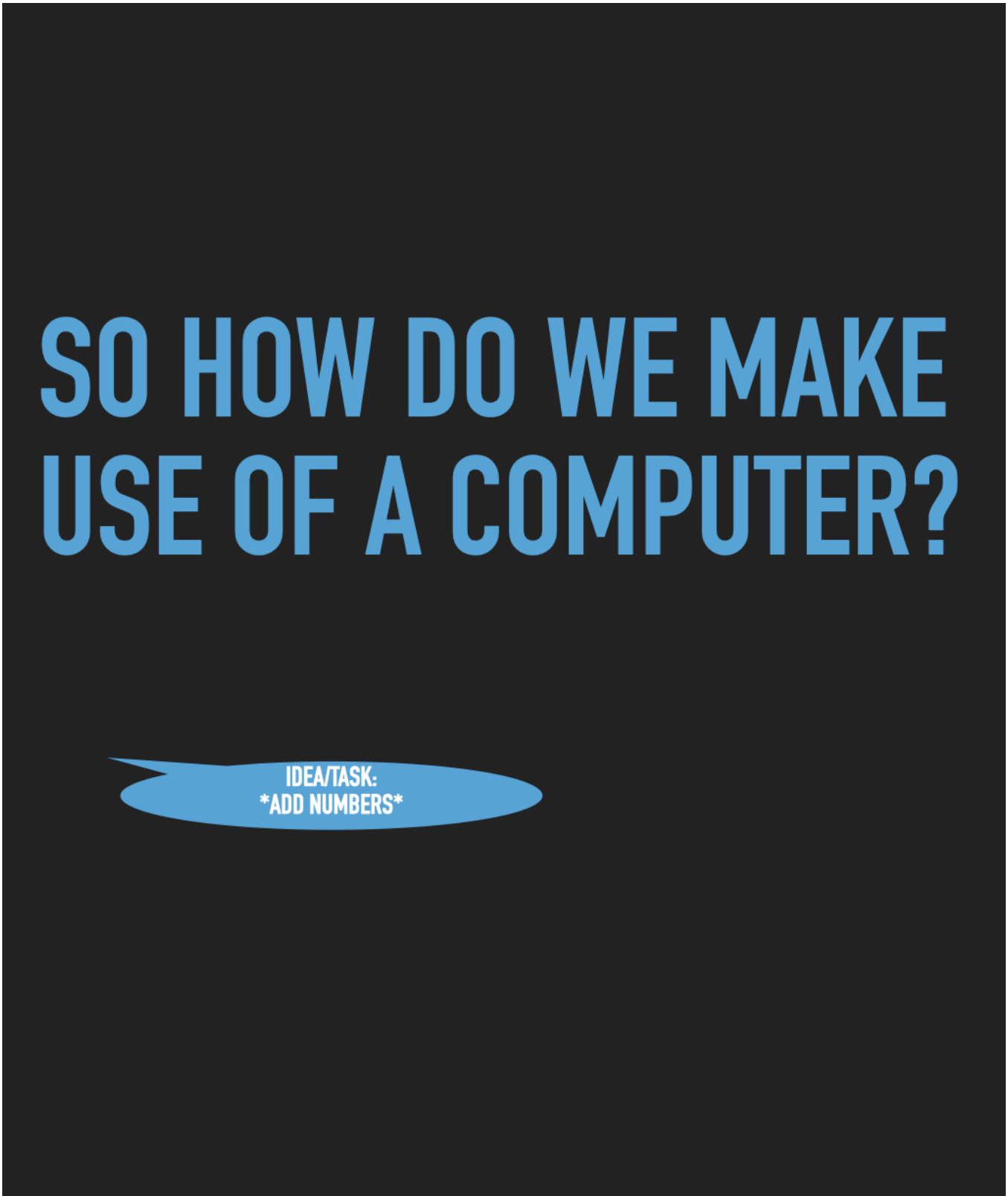
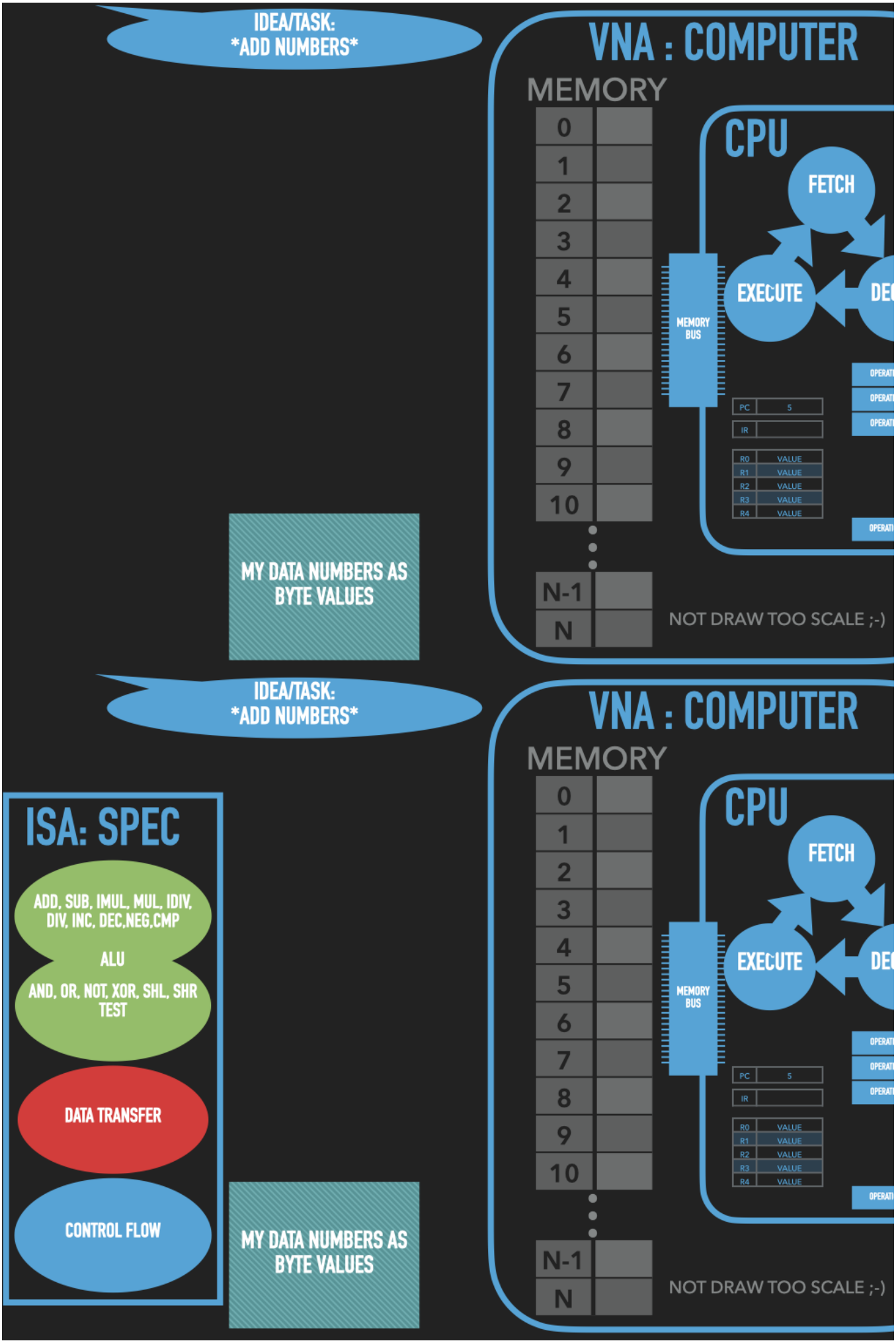
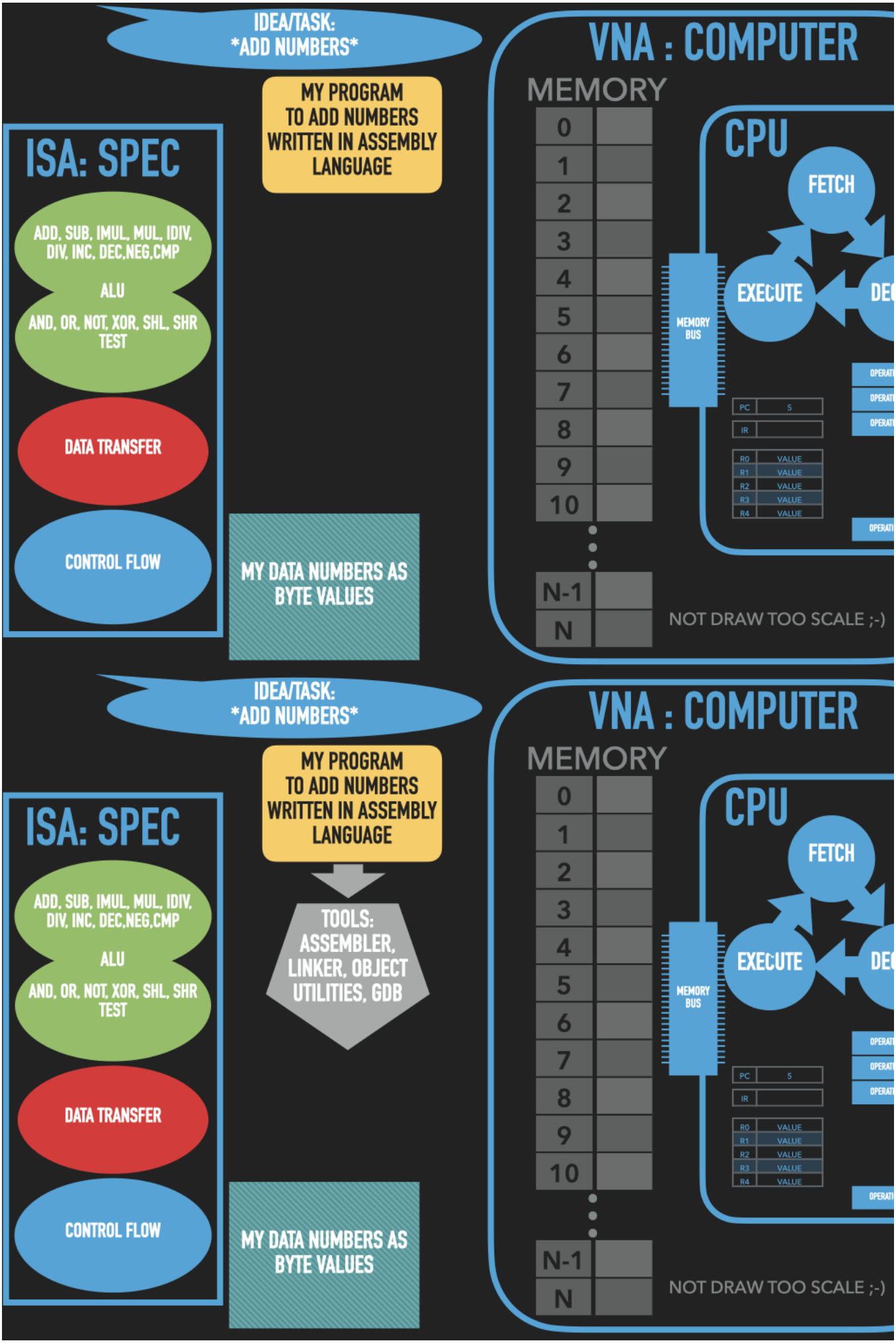
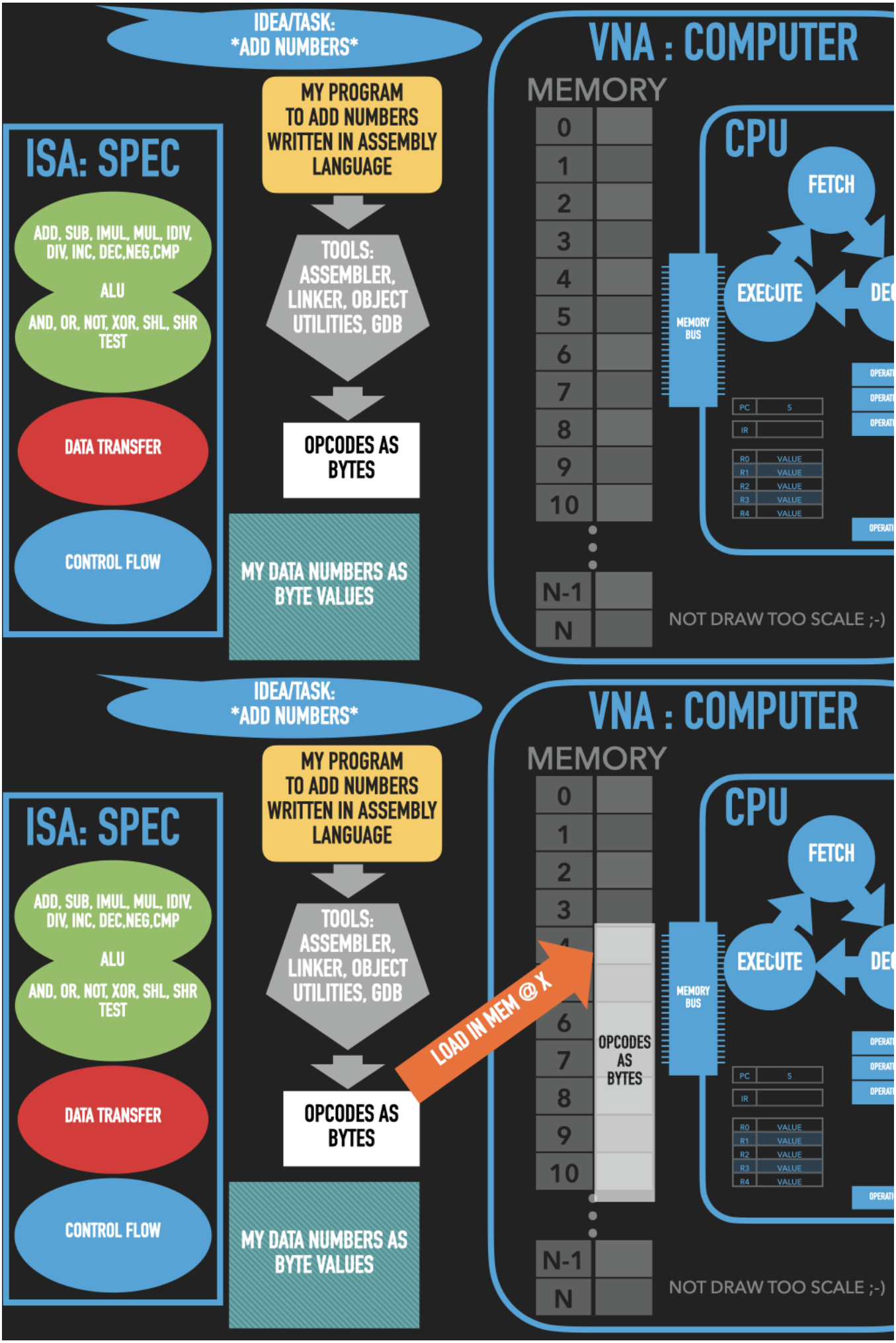
SLS Lecture 10 : Assembly : Program Anat

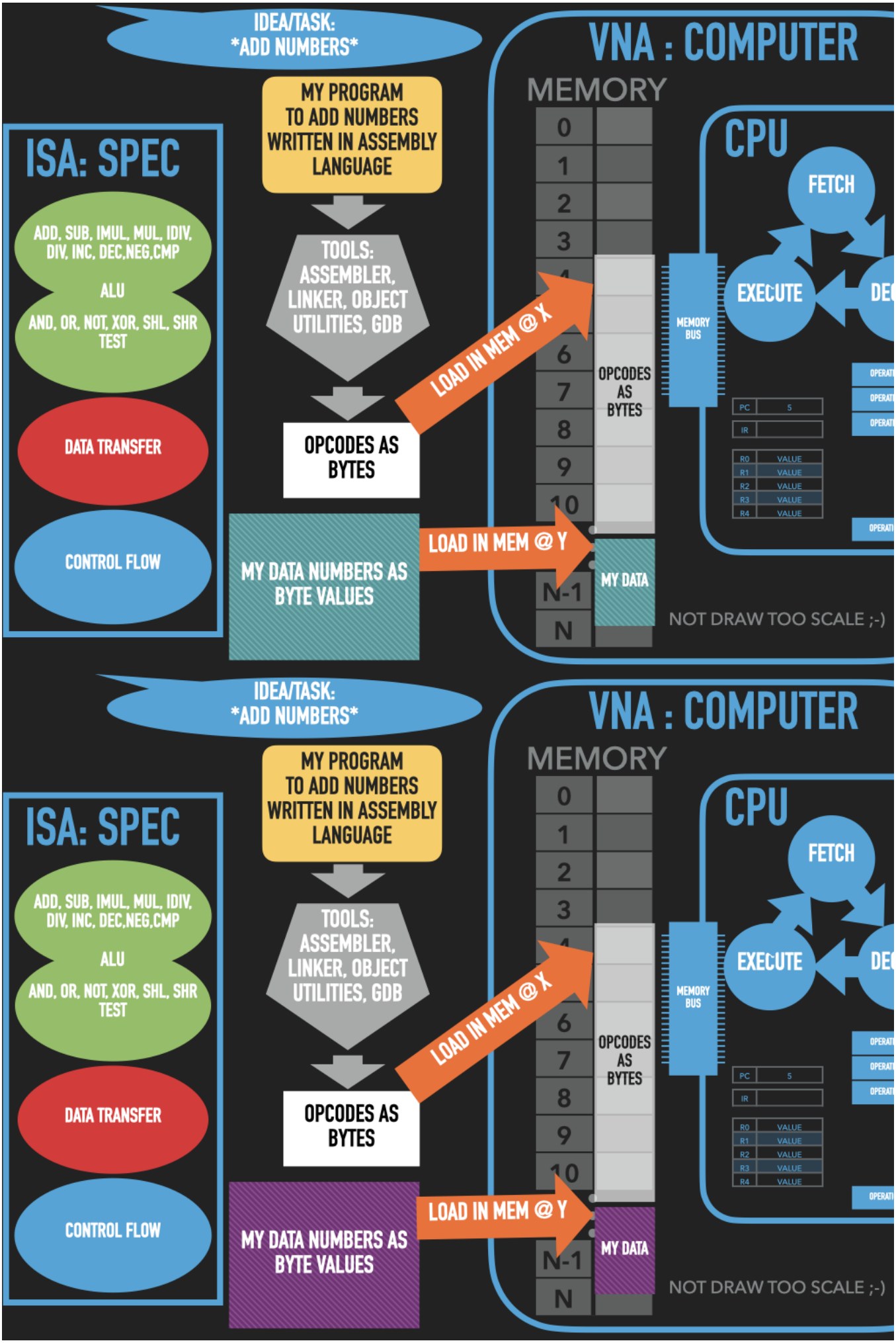
Contents




IDEA/TASK:
ADD NUMBERS
VNA : COMPUTER
MEMORY
0
1
2
3
4
5
6
7
8
9
10
N-1
N
CPU
FETCH
EXECUTE
MEMORY BUS
PC
5
IR
R0 VALUE
R1 VALUE
R2 VALUE
R3 VALUE
R4 VALUE
MY DATA NUMBERS AS BYTE VALUES
NOT DRAW TOO SCALE ;-)
IDEA/TASK:
ADD NUMBERS
VNA : COMPUTER
MEMORY
CPU
FETCH
EXECUTE
MEMORY BUS
PC
5
IR
R0 VALUE
R1 VALUE
R2 VALUE
R3 VALUE
R4 VALUE
ISA: SPEC
ADD, SUB, IMUL, MUL, IDIV, DIV, INC, DEC,NEG,CMP
ALU
AND, OR, NOT, XOR, SHL, SHR TEST
DATA TRANSFER
CONTROL FLOW
MY DATA NUMBERS AS BYTE VALUES
NOT DRAW TOO SCALE ;-)


IDEA/TASK:
ADD NUMBERS
MY PROGRAM
TO ADD NUMBERS
WRITTEN IN ASSEMBLY
LANGUAGE
ISA: SPEC
ADD, SUB, IMUL, MUL, IDIV,
DIV, INC, DEC,NEG,CMP
ALU
AND, OR, NOT, XOR, SHL, SHR
TEST
DATA TRANSFER
CONTROL FLOW
MY DATA NUMBERS AS
BYTE VALUES
VNA : COMPUTER
MEMORY
0
1
2
3
4
5
6
7
8
9
10
N-1
N
CPU
FETCH
EXECUTE
MEMORY
BUS
PC
5
IR
R0 VALUE
R1 VALUE
R2 VALUE
R3 VALUE
R4 VALUE
NOT DRAW TOO SCALE ;-)
IDEA/TASK:
ADD NUMBERS
MY PROGRAM
TO ADD NUMBERS
WRITTEN IN ASSEMBLY
LANGUAGE
TOOLS:
ASSEMBLER,
LINKER, OBJECT
UTILITIES, GDB
ISA: SPEC
ADD, SUB, IMUL, MUL, IDIV,
DIV, INC, DEC,NEG,CMP
ALU
AND, OR, NOT, XOR, SHL, SHR
TEST
DATA TRANSFER
CONTROL FLOW
MY DATA NUMBERS AS
BYTE VALUES
VNA : COMPUTER
MEMORY
CPU
FETCH
EXECUTE
MEMORY
BUS
NOT DRAW TOO SCALE ;-)

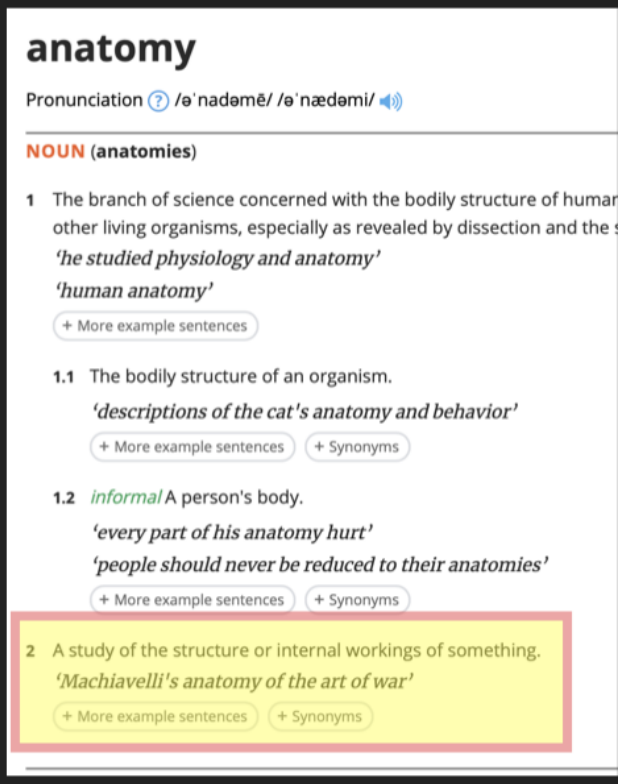

IDEA/TASK:
ADD NUMBERS
MY PROGRAM TO ADD NUMBERS WRITTEN IN ASSEMBLY LANGUAGE
TOOLS: ASSEMBLER, LINKER, OBJECT UTILITIES, GDB
OPCODES AS BYTES
MY DATA NUMBERS AS BYTE VALUES
ISA: SPEC
ADD, SUB, IMUL, MUL, IDIV, DIV, INC, DEC,NEG,CMP
ALU
AND, OR, NOT, XOR, SHL, SHR TEST
DATA TRANSFER
CONTROL FLOW
VNA : COMPUTER
MEMORY
0
1
2
3
4
5
6
7
8
9
10
N-1
N
CPU
FETCH
EXECUTE
MEMORY BUS
PC
5
IR
R0 VALUE
R1 VALUE
R2 VALUE
R3 VALUE
R4 VALUE
NOT DRAW TOO SCALE ;-)
IDEA/TASK:
ADD NUMBERS
MY PROGRAM TO ADD NUMBERS WRITTEN IN ASSEMBLY LANGUAGE
TOOLS: ASSEMBLER, LINKER, OBJECT UTILITIES, GDB
OPCODES AS BYTES
LOAD IN MEM @ X
MY DATA NUMBERS AS BYTE VALUES
ISA: SPEC
ADD, SUB, IMUL, MUL, IDIV, DIV, INC, DEC,NEG,CMP
ALU
AND, OR, NOT, XOR, SHL, SHR TEST
DATA TRANSFER
CONTROL FLOW
VNA : COMPUTER
MEMORY
OPCODES AS BYTES
CPU
FETCH
EXECUTE
MEMORY BUS
NOT DRAW TOO SCALE ;-)


IDEA/TASK:
ADD NUMBERS
MY PROGRAM TO ADD NUMBERS WRITTEN IN ASSEMBLY LANGUAGE
TOOLS: ASSEMBLER, LINKER, OBJECT UTILITIES, GDB
OPCODES AS BYTES
MY DATA NUMBERS AS BYTE VALUES
LOAD IN MEM @ X
LOAD IN MEM @ Y
ISA: SPEC
ADD, SUB, IMUL, MUL, IDIV, DIV, INC, DEC,NEG,CMP
ALU
AND, OR, NOT, XOR, SHL, SHR TEST
DATA TRANSFER
CONTROL FLOW
VNA : COMPUTER
MEMORY
0
1
2
3
6
7
8
9
10
N-1
N
OPCODES AS BYTES
MY DATA
MEMORY BUS
CPU
FETCH
EXECUTE
PC
5
IR
R0 VALUE
R1 VALUE
R2 VALUE
R3 VALUE
R4 VALUE
NOT DRAW TOO SCALE ;-)
IDEA/TASK:
ADD NUMBERS
MY PROGRAM TO ADD NUMBERS WRITTEN IN ASSEMBLY LANGUAGE
TOOLS: ASSEMBLER, LINKER, OBJECT UTILITIES, GDB
OPCODES AS BYTES
MY DATA NUMBERS AS BYTE VALUES
LOAD IN MEM @ X
LOAD IN MEM @ Y
ISA: SPEC
ADD, SUB, IMUL, MUL, IDIV, DIV, INC, DEC,NEG,CMP
ALU
AND, OR, NOT, XOR, SHL, SHR TEST
DATA TRANSFER
CONTROL FLOW
VNA : COMPUTER
MEMORY
0
1
2
3
6
7
8
9
10
N-1
N
OPCODES AS BYTES
MY DATA
MEMORY BUS
CPU
FETCH
EXECUTE
PC
5
IR
R0 VALUE
R1 VALUE
R2 VALUE
R3 VALUE
R4 VALUE
NOT DRAW TOO SCALE ;-)

USING THE EXAMPLE WE ARE GOING TO WORK THROUGH

THE ANATOMY OF A PROGRAM

- Simple linear sequence of instructions
- Repeat sequence
  - Go back to a prior location in a sequence – special case of goto
  - Goto an arbitrary place
- Conditional
  - If goto
  - If mov
- Advanced :
  - Functions & recursion
  - Jump table



anatomy

Pronunciation /əˈnadəmē/ /əˈnædəmi/

NOUN (**anatomies**)

1 The branch of science concerned with the bodily structure of humar other living organisms, especially as revealed by dissection and the
'he studied physiology and anatomy'
'human anatomy'
+ More example sentences

1.1 The bodily structure of an organism.
'descriptions of the cat's anatomy and behavior'
+ More example sentences + Synonyms

1.2 *informal* A person's body.
'every part of his anatomy hurt'
'people should never be reduced to their anatomies'
+ More example sentences + Synonyms

2 A study of the structure or internal workings of something.
'Machiavelli's anatomy of the art of war'
+ More example sentences + Synonyms

BEFORE WE GET GOING WHAT ARE WE MISSING?

$$S = \sum_{i=0}^{n} x_i$$

VARIABLES AND DATA STRUCTURES

```
int i;
for (i=0; i<n; i++) {
  S = S + x[i]
}
```

SCALARS, ARRAYS, LISTS, ETC …

VARIABLES, DATA STRUCTURES AND INPUT DATA : ORGANIZED BYTES IN MEMORY

LOCATIONS AND VALUES

- We pick locations to hold our data:
- We pick the number of bytes we need to reserve based on what we want data we plan to store and work with
- Our job to stay organized and keep track of things
- Write code that moves data in and out of registers so that we can do operations on it

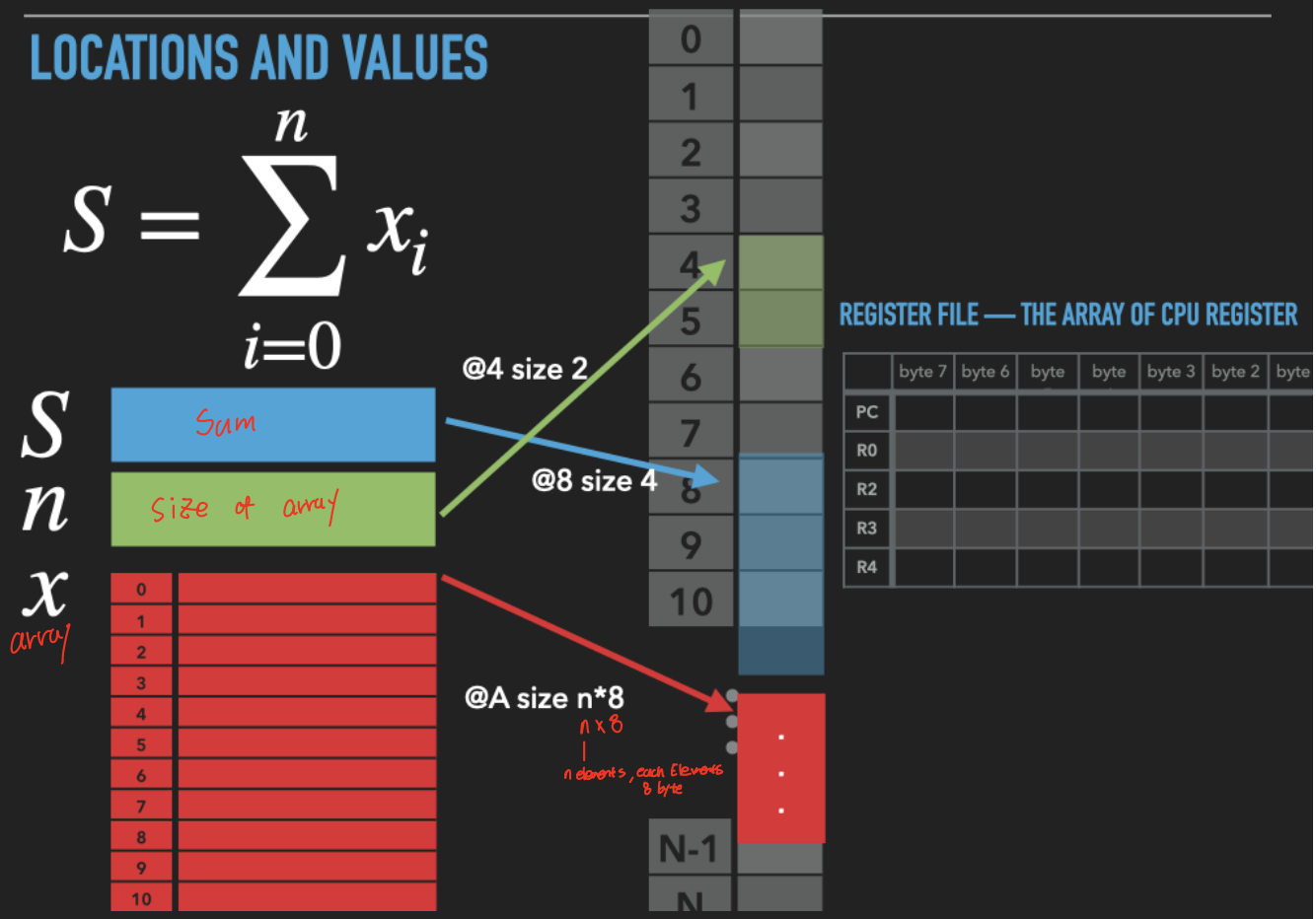
REGISTER FILE — THE ARRAY OF CPU REGISTER

	byte 7	byte 6	byte	byte	byte 3	byte 2	byte
PC							
R0							
R2							
R3							
R4							

VARIABLES, DATA STRUCTURES AND INPUT DATA : ORGANIZED BYTES IN MEMORY

LOCATIONS AND VALUES

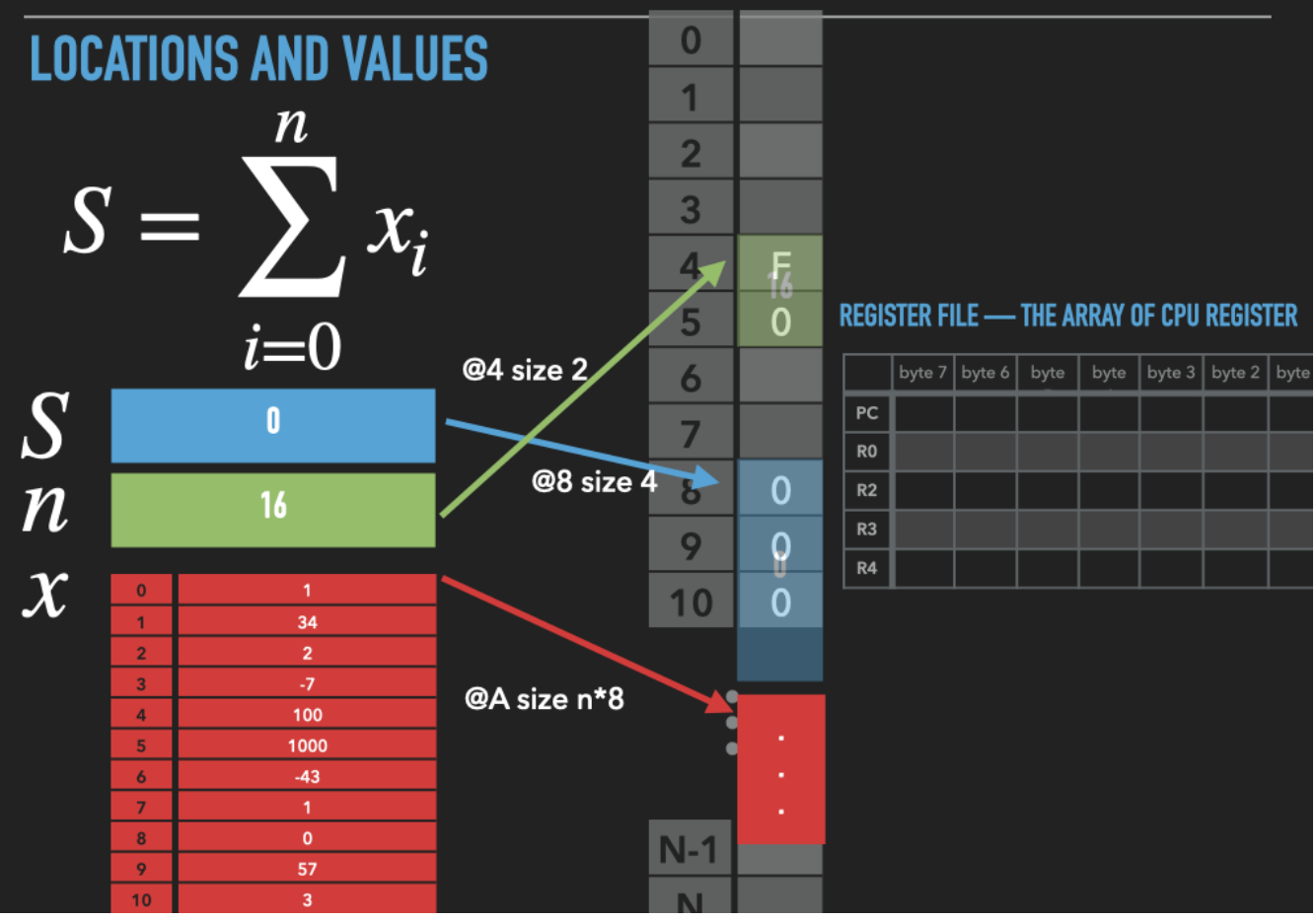
$$S = \sum_{i=0}^{n} x_i$$

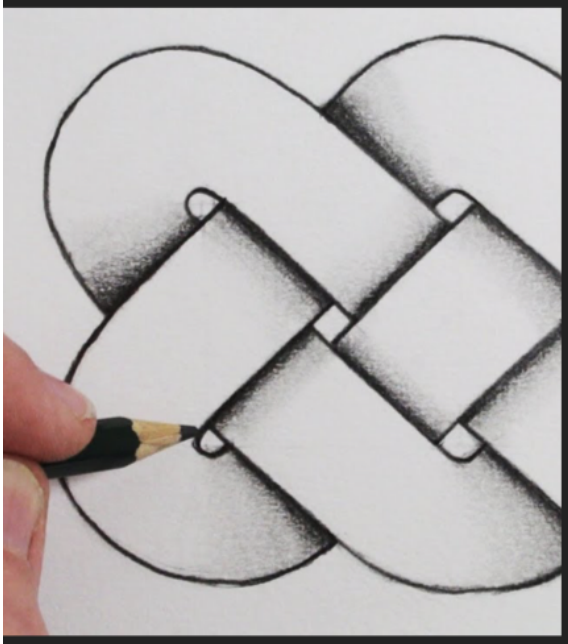


VARIABLES, DATA STRUCTURES AND INPUT DATA : ORGANIZED BYTES IN MEMORY

LOCATIONS AND VALUES

$$S = \sum_{i=0}^{n} x_i$$



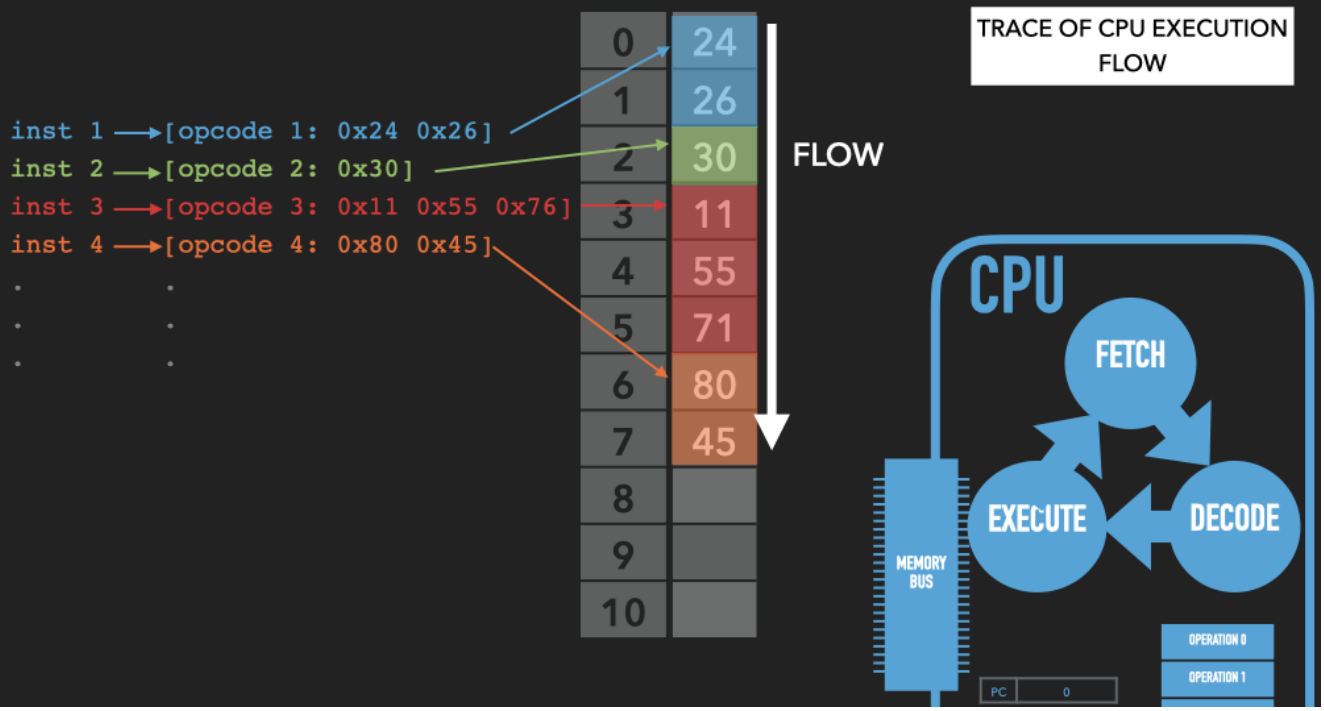


LEARN TO DRAW & VISUALIZE YOUR PROGRAMS

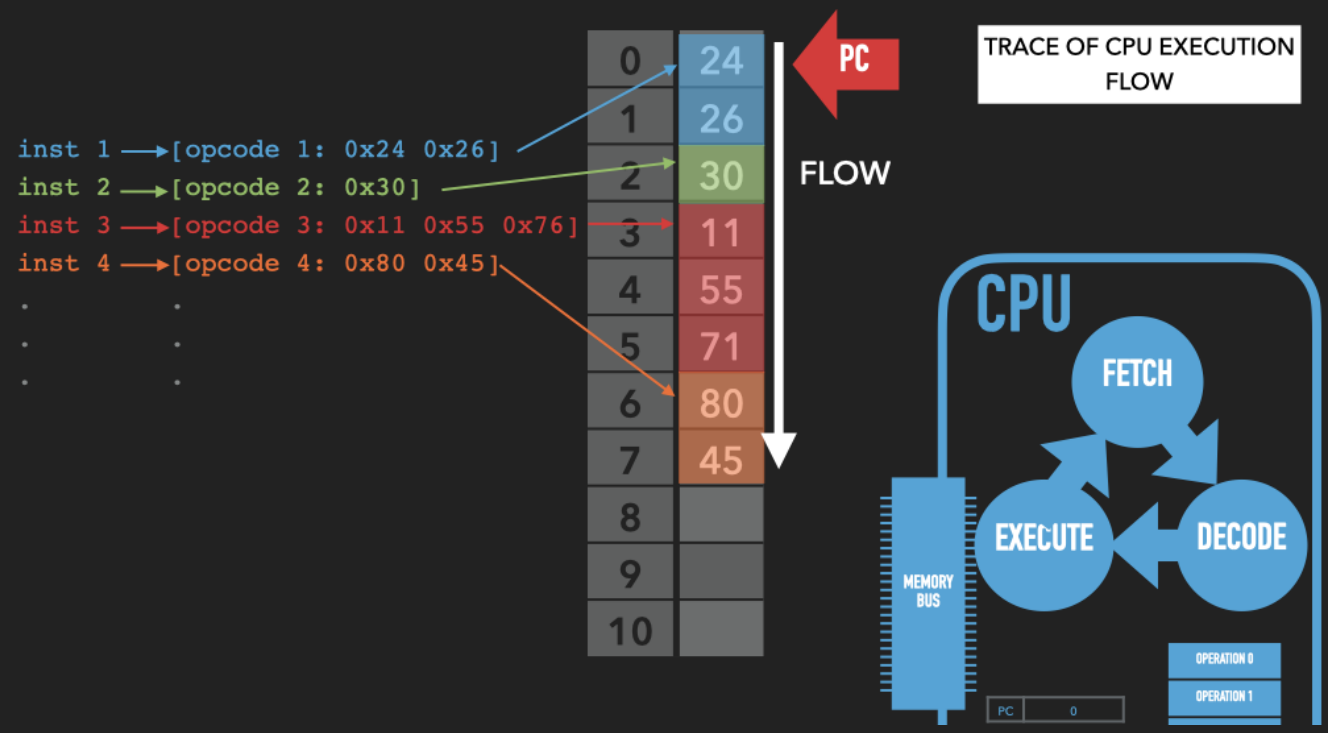
EVERYTHING MUST BE REPRESENTED IN MEMORY AND REGISTERS

LOCATION AND SIZE: EVERYTHING HAS A SHAPE AND MOST OFTEN ARROWS CONNECTING THINGS

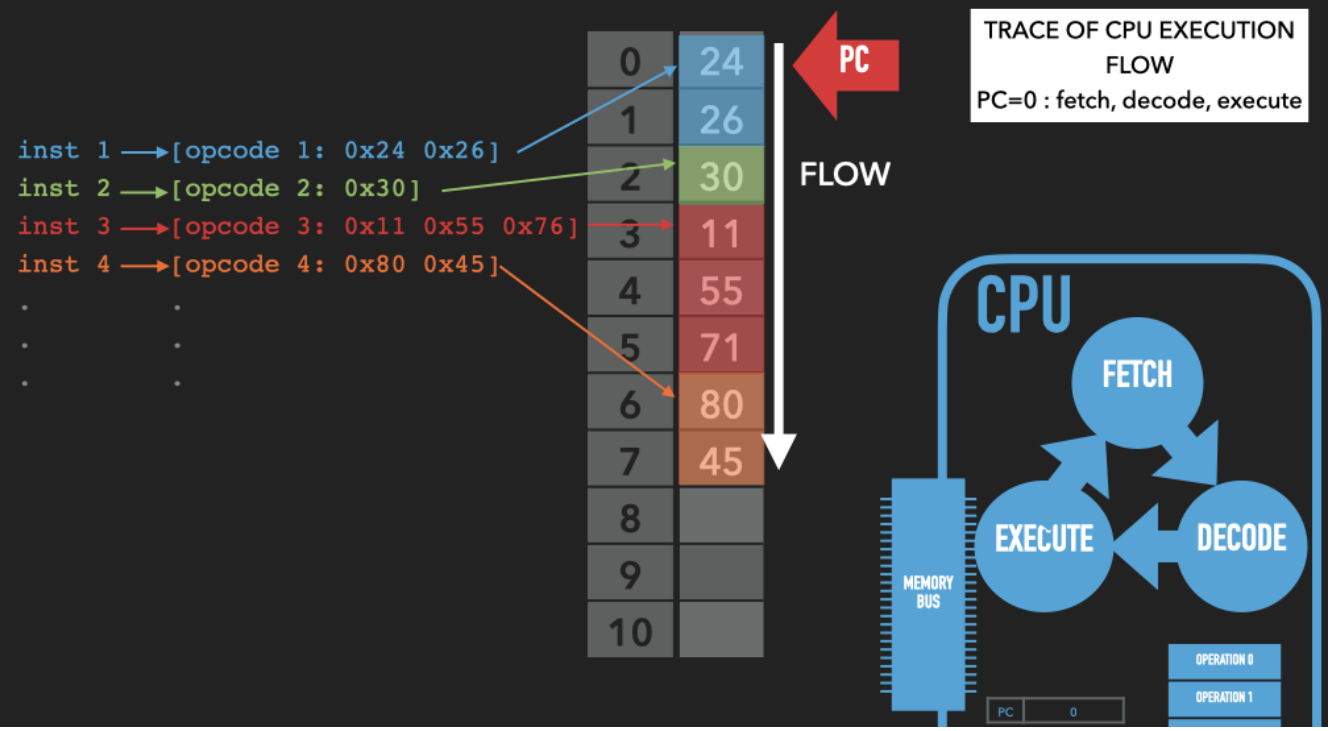
SIMPLE SEQUENCE:
— LINEAR LAYOUT
— LINEAR "FLOW" OF EXECUTION



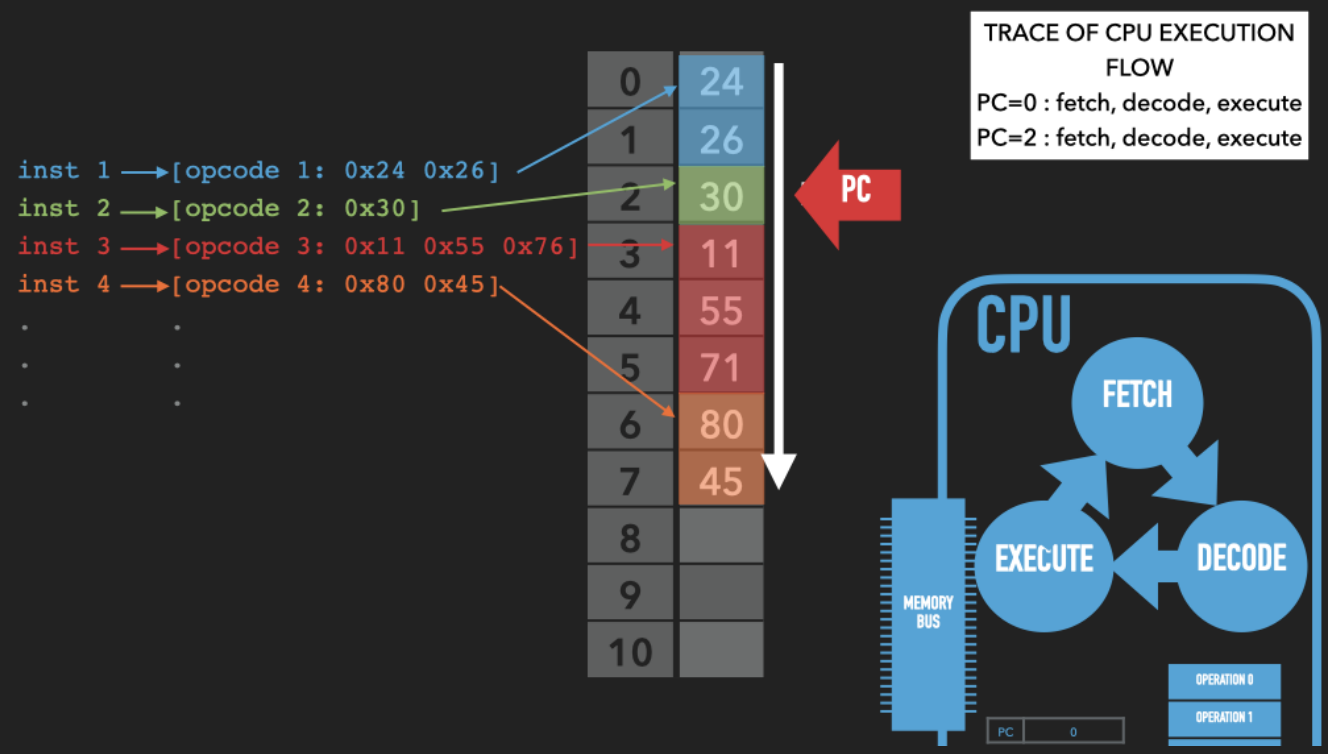
SIMPLE SEQUENCE:
— LINEAR LAYOUT
— LINEAR "FLOW" OF EXECUTION



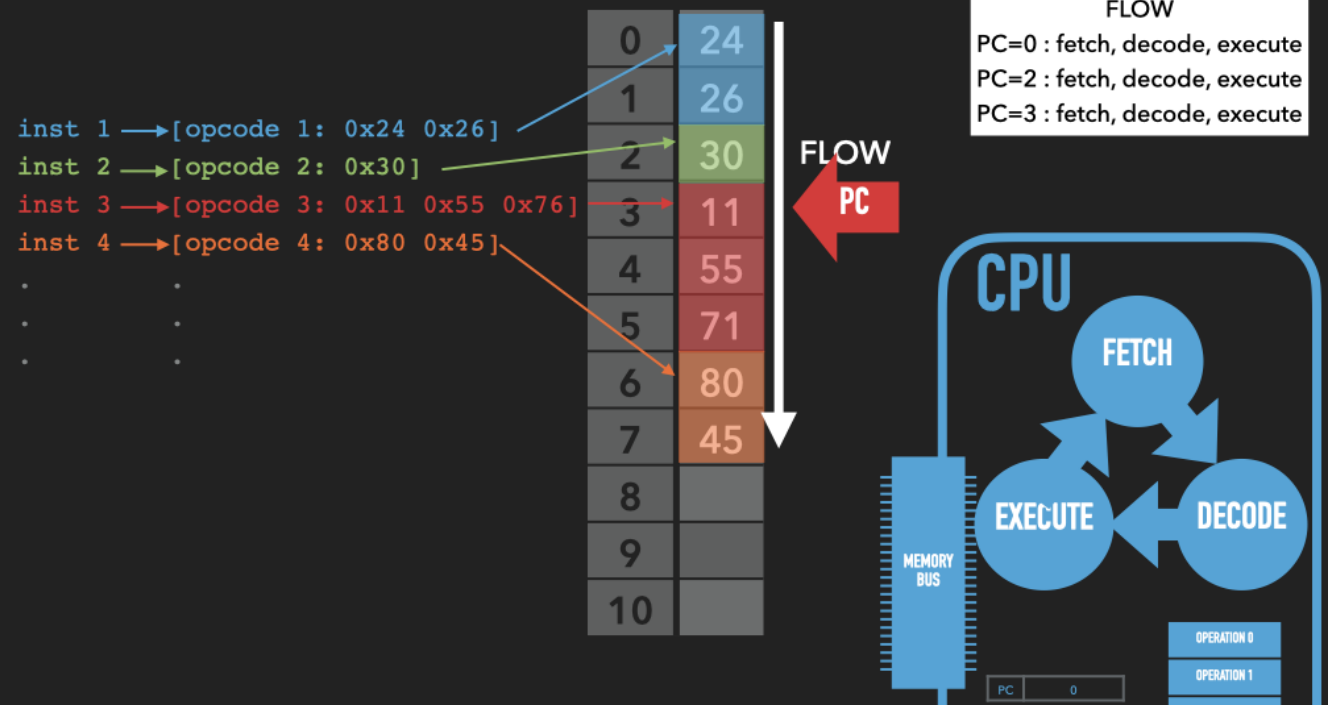
SIMPLE SEQUENCE:
— LINEAR LAYOUT
— LINEAR "FLOW" OF EXECUTION



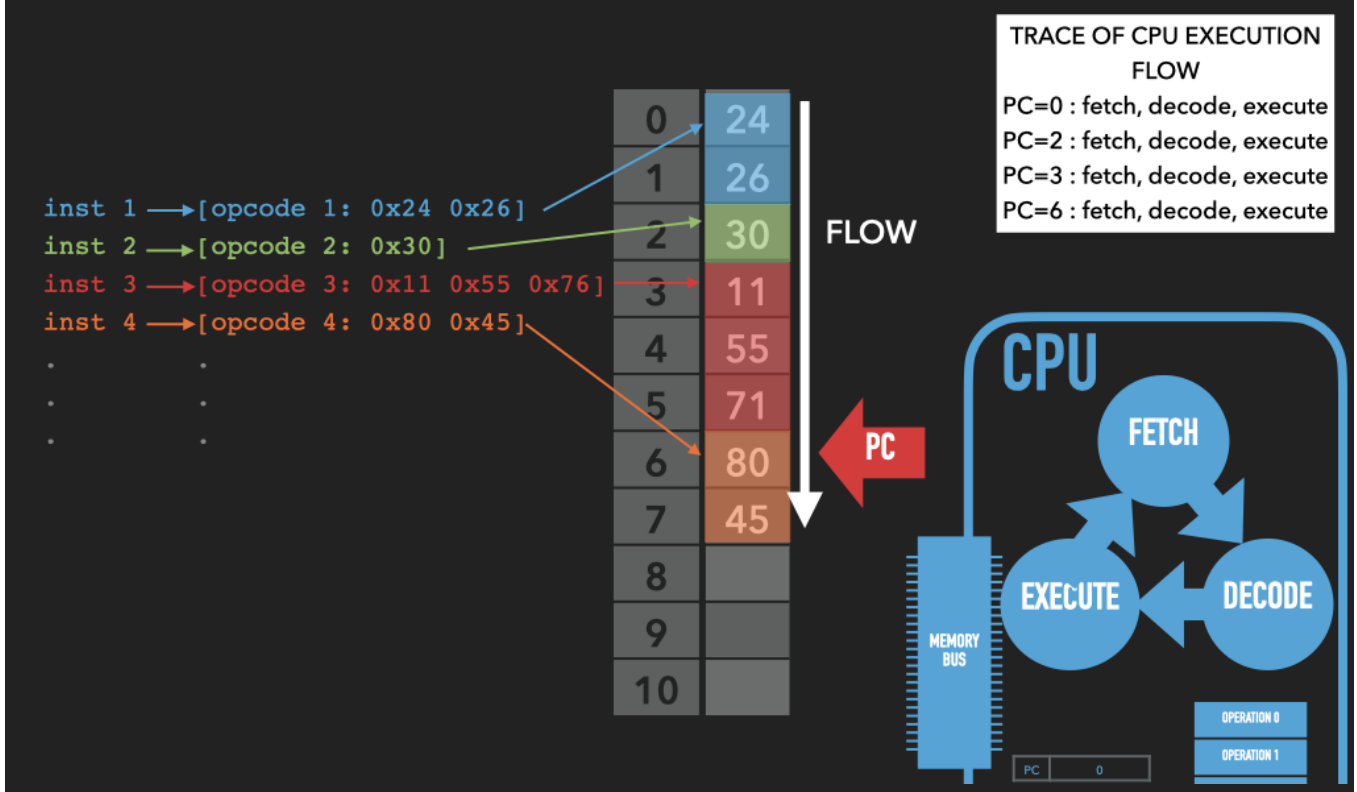
SIMPLE SEQUENCE:
— LINEAR LAYOUT
— LINEAR "FLOW" OF EXECUTION



SIMPLE SEQUENCE:
— LINEAR LAYOUT
— LINEAR "FLOW" OF EXECUTION

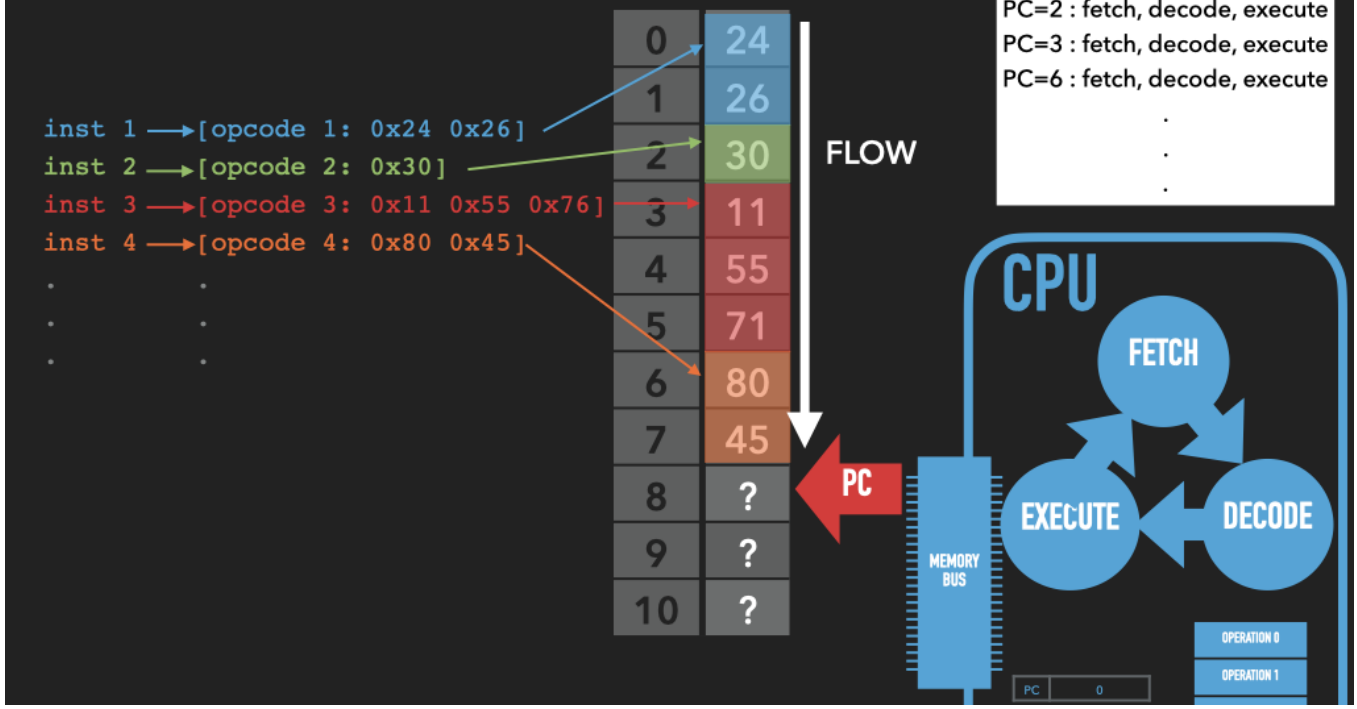


SIMPLE SEQUENCE:
— LINEAR LAYOUT
— LINEAR "FLOW" OF EXECUTION



TRACE OF CPU EXECUTION FLOW
PC=0 : fetch, decode, execute
PC=2 : fetch, decode, execute
PC=3 : fetch, decode, execute
PC=6 : fetch, decode, execute

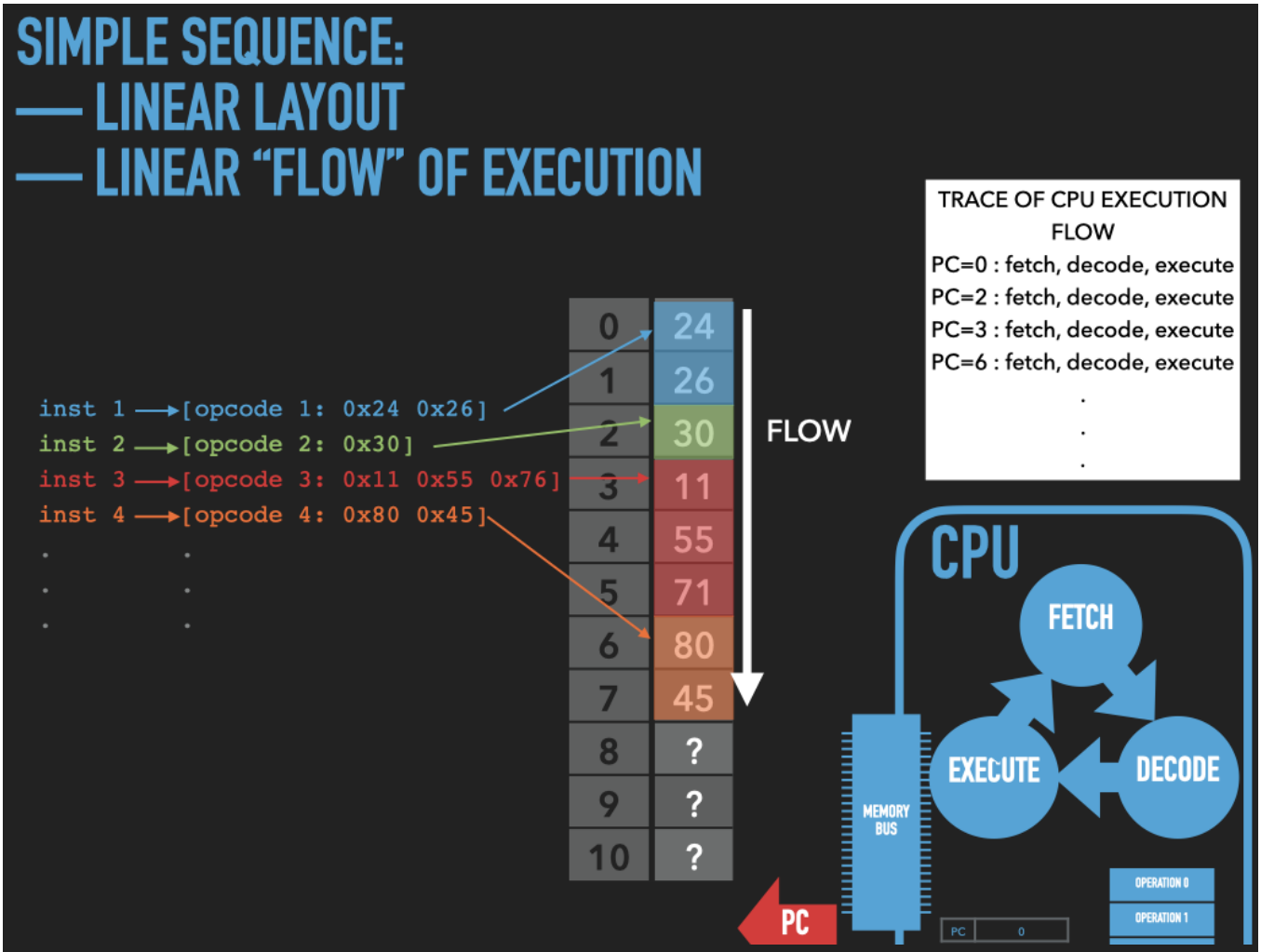
SIMPLE SEQUENCE:
— LINEAR LAYOUT
— LINEAR "FLOW" OF EXECUTION



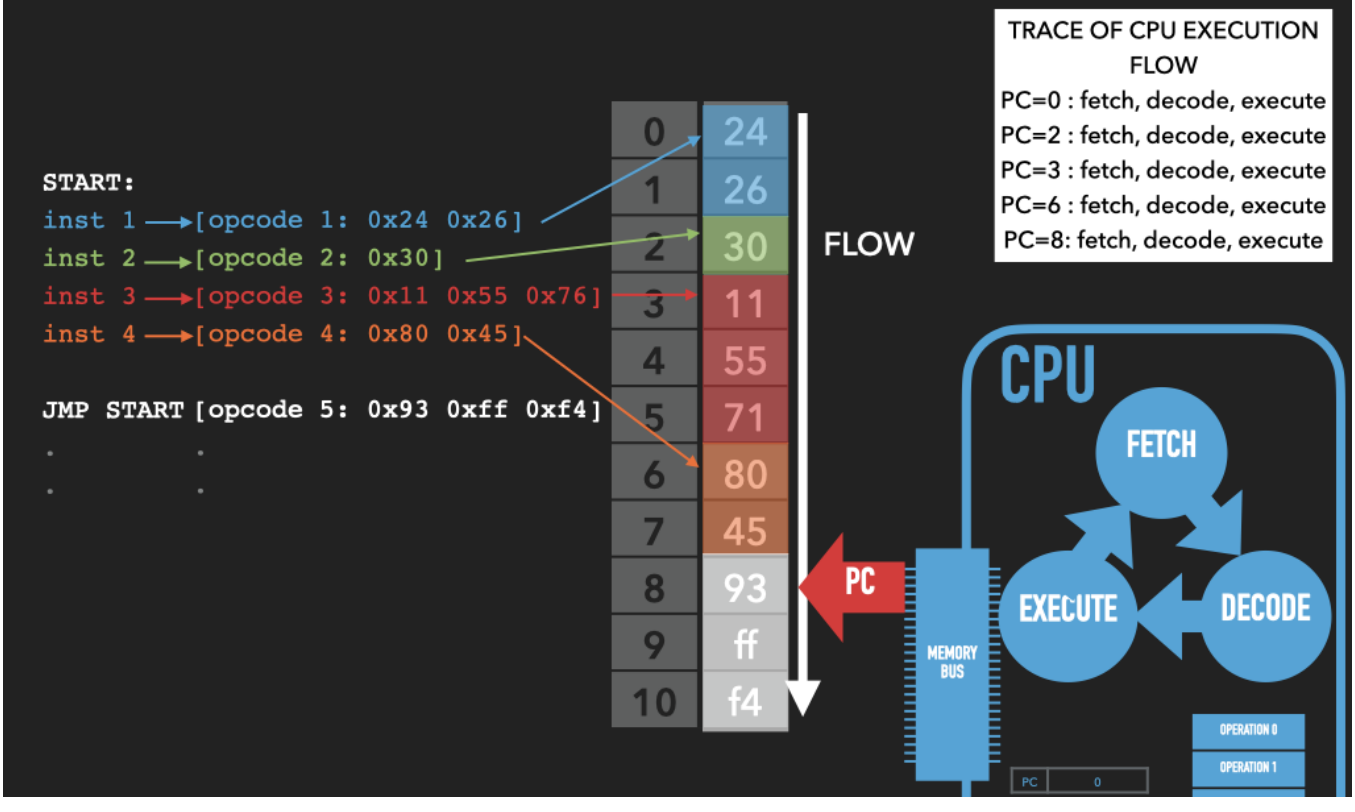
TRACE OF CPU EXECUTION FLOW
PC=0 : fetch, decode, execute
PC=2 : fetch, decode, execute
PC=3 : fetch, decode, execute
PC=6 : fetch, decode, execute

.
.
.

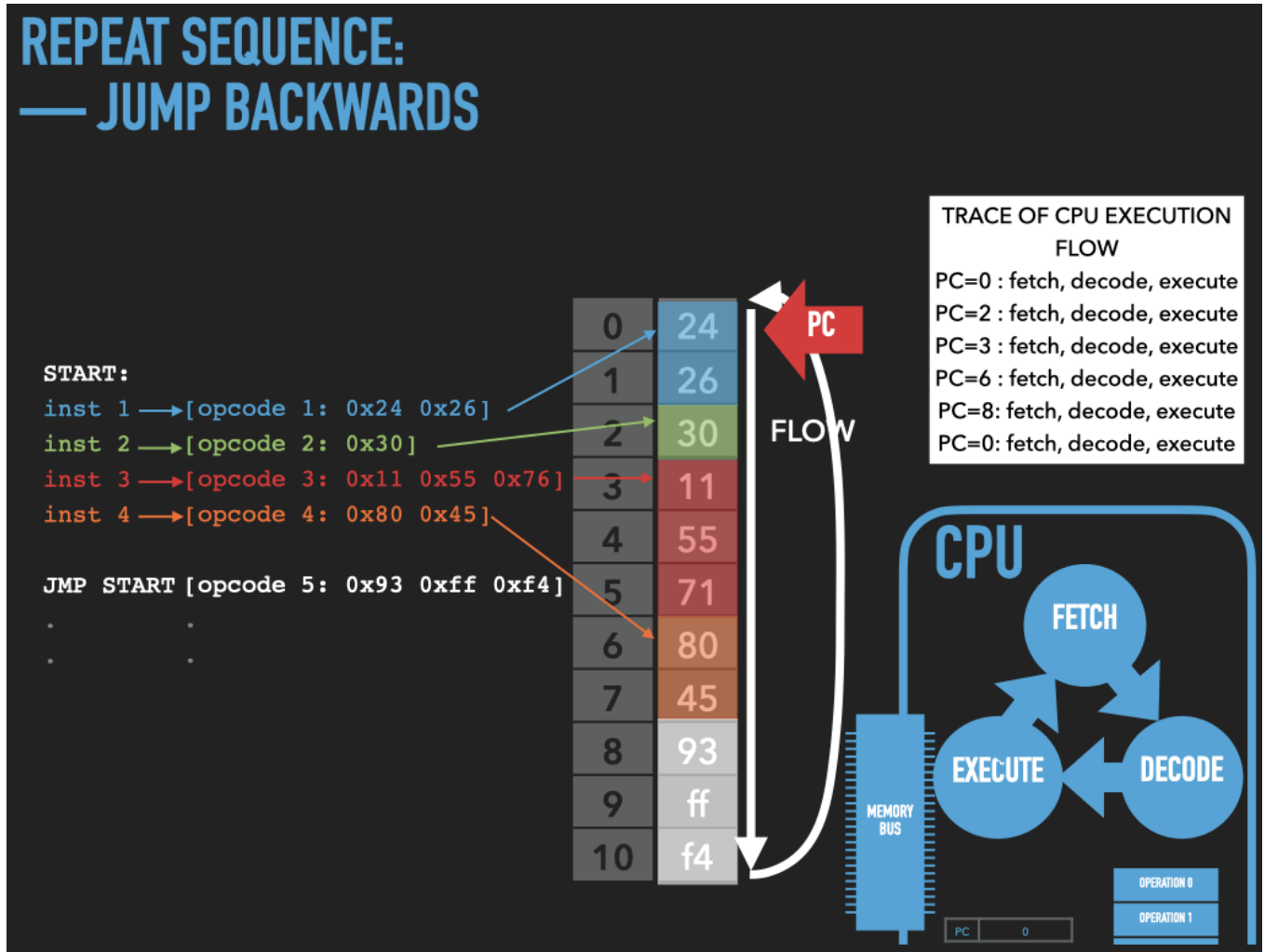
SIMPLE SEQUENCE:
— LINEAR LAYOUT
— LINEAR "FLOW" OF EXECUTION



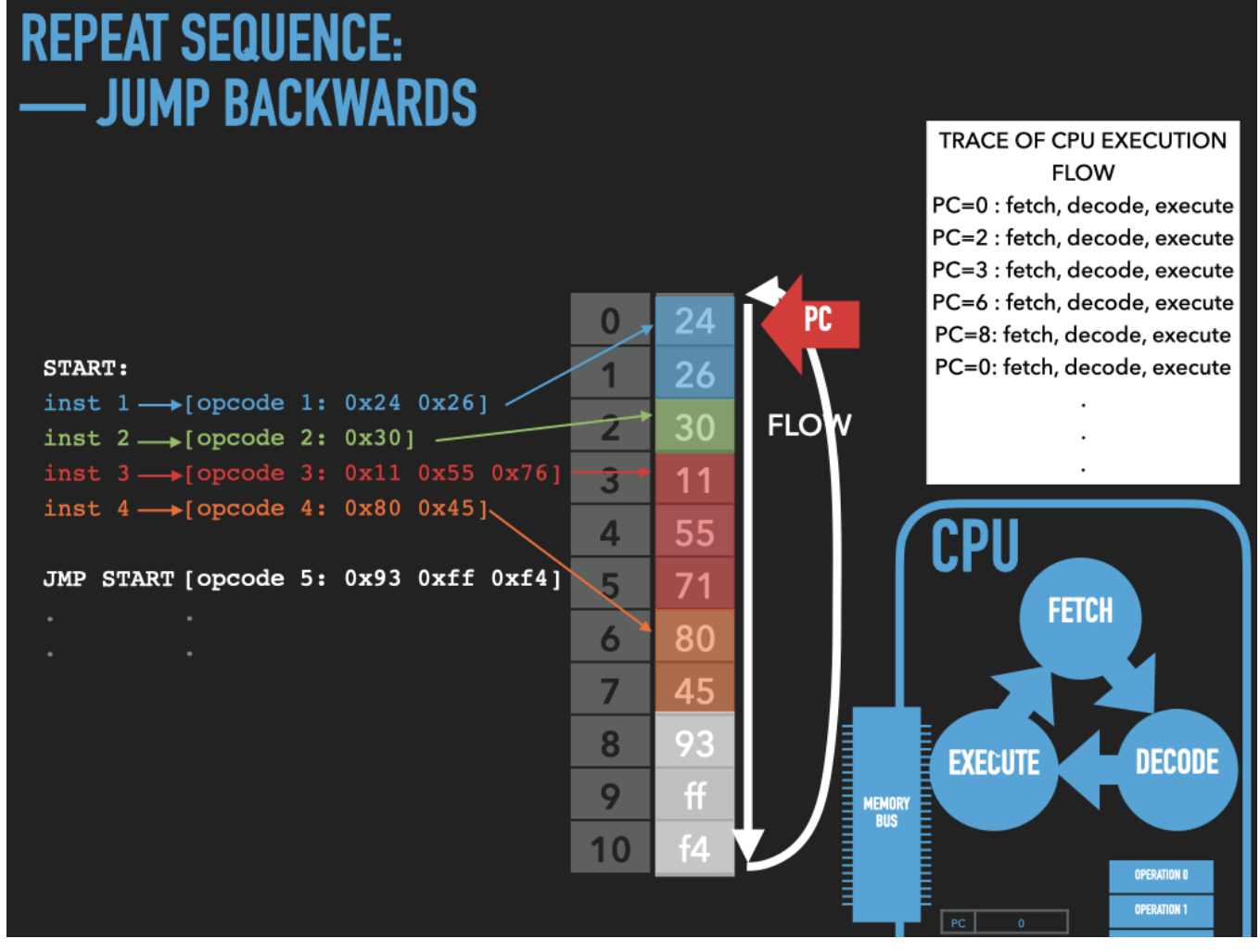
REPEAT SEQUENCE:
— JUMP BACKWARDS



REPEAT SEQUENCE:
— JUMP BACKWARDS



REPEAT SEQUENCE:
— JUMP BACKWARDS



CONDITIONS AND IF
— FORK FLOW : A VS B
— MOVE IF : CONDITIONAL MOVE

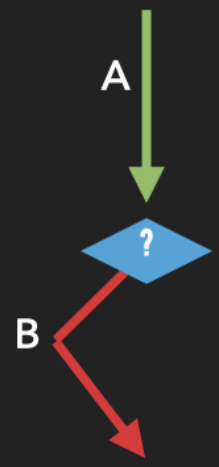
```
A
if ? then
   B
else
   C
D
```

CONDITIONS AND IF
— FORK FLOW : A VS B


```
A
if ? then
   B
else
   C
D
```

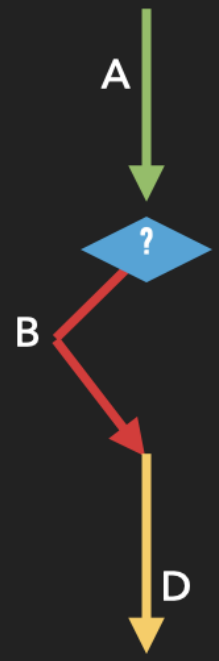
CONDITIONS AND IF
— FORK FLOW : A VS B

A
if ? then
  B
else
  C
D



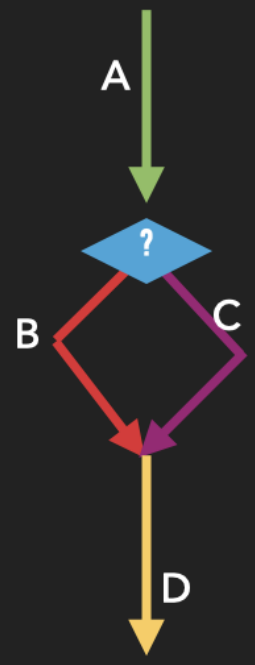
CONDITIONS AND IF
— FORK FLOW : A VS B

A
if ? then
  B
else
  C
D



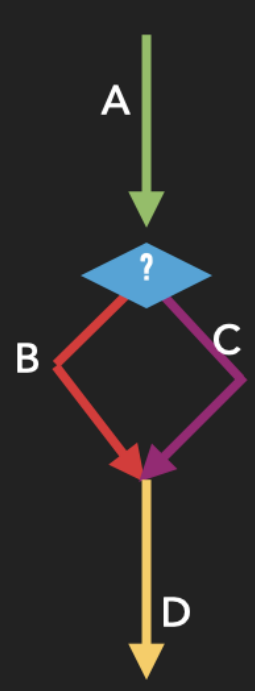
CONDITIONS AND IF — FORK FLOW : A VS B

A
if ? then
  B
else
  C
D



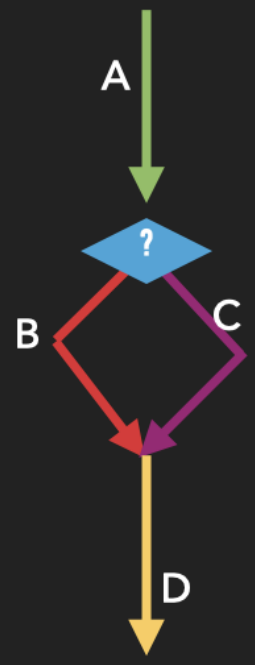
CONDITIONS AND IF — FORK FLOW : A VS B

A
if ? then
  B
else
  C
D



CONDITIONS AND IF
— FORK FLOW : A VS B

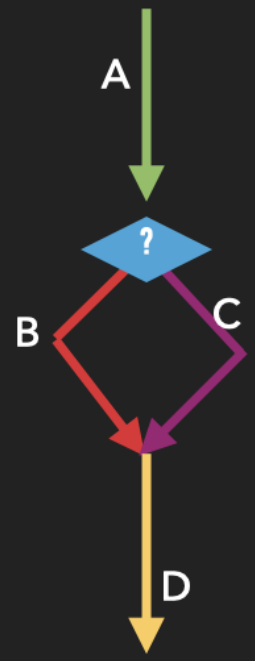
A
if ? then
B
else
C
D



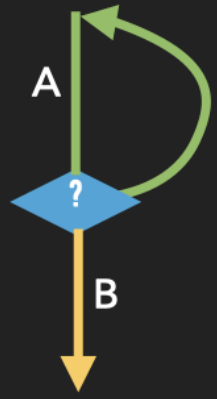
do
A
while ?
B

CONDITIONS AND IF
— FORK FLOW : A VS B

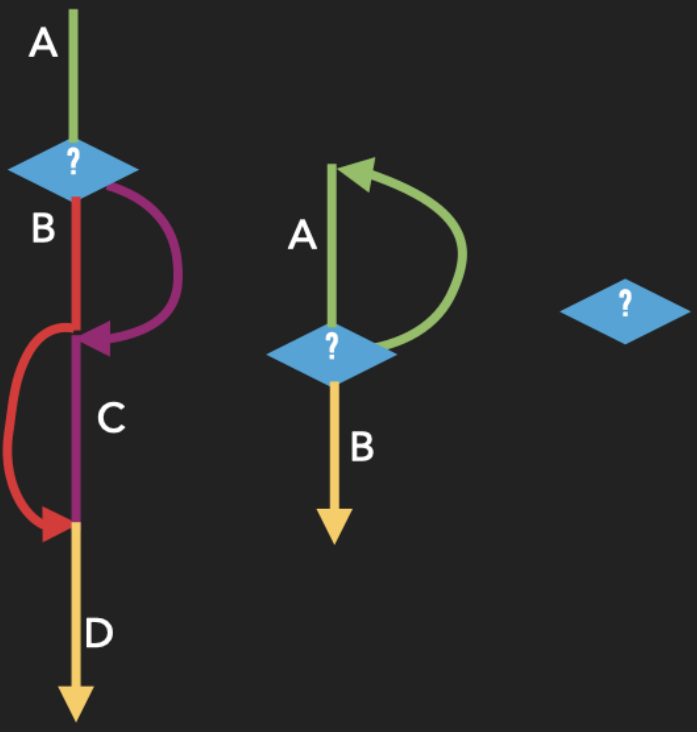
A
if ? then
B
else
C
D



do
A
while ?
B



IF CONDITION THEN-> 1) CONDITION 2) GOTO ELSE FALL THROUGH



- We implement with a pair of instructions that cooperate with a special set of single bit "condition" flags/registers

1. AN ALU INSTRUCTION

2. A CONDITIONAL JUMP INSTRUCTION

TEST,AND,CMP,SUB
J<CC>

IF CONDITION THEN-> 1) CONDITION 2) GOTO ELSE FALL THROUGH

- We implement with a pair of instructions that cooperate with a special set of single bit "condition" flags/registers

CONDITION FLAGS : BIT FIELDS OF EFLAGS

CF overflow	OF signed overflow	SF output is negative Sign flag	ZF output is zero
0	0	0	0

jl

PC

1. AN ALU INSTRUCTION

2. A CONDITIONAL JUMP INSTRUCTION

- ALU : INSTRUCTIONS DO SOME EXTRA WORK: AFTER RESULT IS CALCULATE THEY WILL SET CONDITION FLAGS: EG. SUB – RAX, RBX –> RBX = RBX - RAX AND SET OF, SF, ZF, AF, PF, and :CF flags are set according to the result

IF CONDITION THEN-> 1) CONDITION 2) GOTO ELSE FALL THROUGH

- We implement with a pair of instructions that cooperate with a special set of single bit "condition" flags/registers

CONDITION FLAGS : BIT FIELDS OF EFLAGS

CF	OF	SF	ZF
0	0	0	1

RBX = RAX then 1

PC

```
1. SUB RBX, RAX

2. A CONDITIONAL
   JUMP INSTRUCTION
```

- ALU : INSTRUCTIONS DO SOME EXTRA WORK: AFTER RESULT IS CALCULATE THEY WILL SET CONDITION FLAGS: EG. SUB RBX, RBA –> RBX = RBX - RAX AND SET OF, SF, ZF, AF, PF, and :CF flags are set according to the result

IF CONDITION THEN-> 1) CONDITION 2) GOTO ELSE FALL THROUGH

- We implement with a pair of instructions that cooperate with a special set of single bit "condition" flags/registers

CONDITION FLAGS : BIT FIELDS OF EFLAGS

CF	OF	SF	ZF
0	0	0	1

PC

```
1. SUB RBX, RAX

2. A CONDITIONAL
   JUMP INSTRUCTION
```

check whichever flag

- Conditional Jump instructions : j<cc> jump (change the pc) **if and only if** a certain set of condition flags are set otherwise the pc is updated to the address of the opcode following the jump instruction in linear order eg we skip doing the jump

condition code

- **see manual for the full list there are a lot of them!**

IF CONDITION THEN-> 1) CONDITION 2) GOTO ELSE FALL THROUGH

- We implement with a pair of instructions that cooperate with a special set of single bit "condition" flags/registers

CONDITION FLAGS : BIT FIELDS OF EFLAGS

CF	OF	SF	ZF
0	0	0	1

PC

```
1. SUB RBX, RAX
2. JE <ADDR>
```

- eg. JE – "Jump ~~near~~ if equal (ZF=1)" -- PC = <ADDR> else PC continues to address of next opcode as "usual"
  - So that's it that's our "fork in execution flow"
  - JZ is equivalent
- similarly JNE - "Jump near if not equal (ZF=0)" : JNZ equivalent

CONDITIONS AND IF — MOV IF

if ? {x=a} else no change to x

CONDITION FLAGS : BIT FIELDS OF EFLAGS

CF	OF	SF	ZF
0	0	0	1

```
1. SUB RBX, RAX
2. CMOVEQ <ADDR>
```

- "Move if equal (ZF=1)" else **DO NOTHING**. Similarly CMOVNE "Move if not equal (ZF=0). Remember Q just tells us the "size" of the mov

TEXT

ADD, SUB, IMUL, MUL, IDIV, DIV, INC, DEC,NEG,CMP

ALU

AND, OR, NOT, XOR, SHL, SHR TEST

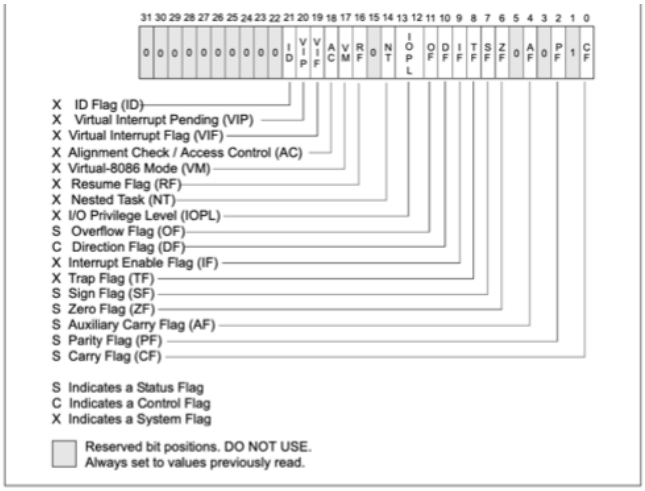


Figure 3-8. EFLAGS Register

As the IA-32 Architecture has evolved, flags have been added to the EFLAGS register, but the function and placement of existing flags have remained the same from one family of the IA-32 processors to the next. As a result, code that accesses or modifies these flags for one family of IA-32 processors works as expected when run on later families of processors.

3.4.3.1 Status Flags

The status flags (bits 0, 2, 4, 6, 7, and 11) of the EFLAGS register indicate the results of arithmetic instructions, such as the ADD, SUB, MUL, and DIV instructions. The status flag functions are:

CF (bit 0) **Carry flag** — Set if an arithmetic operation generates a carry or a borrow out of the most-significant bit of the result; cleared otherwise. This flag indicates an overflow condition for unsigned-integer arithmetic. It is also used in multiple-precision arithmetic.

PF (bit 2) **Parity flag** — Set if the least-significant byte of the result contains an even number of 1 bits; cleared otherwise.

AF (bit 4) **Auxiliary Carry flag** — Set if an arithmetic operation generates a carry or a borrow out of bit 3 of the result; cleared otherwise. This flag is used in binary-coded decimal (BCD) arithmetic.

ZF (bit 6) **Zero flag** — Set if the result is zero; cleared otherwise.

SF (bit 7) **Sign flag** — Set equal to the most-significant bit of the result, which is the sign bit of a signed integer. (0 indicates a positive value and 1 indicates a negative value.)

OF (bit 11) **Overflow flag** — Set if the integer result is too large a positive number or too small a negative number (excluding the sign-bit) to fit in the destination operand; cleared otherwise. This flag indicates an overflow condition for signed-integer (two's complement) arithmetic.

Of these status flags, only the CF flag can be modified directly, using the STC, CLC, and CMC instructions. Also the

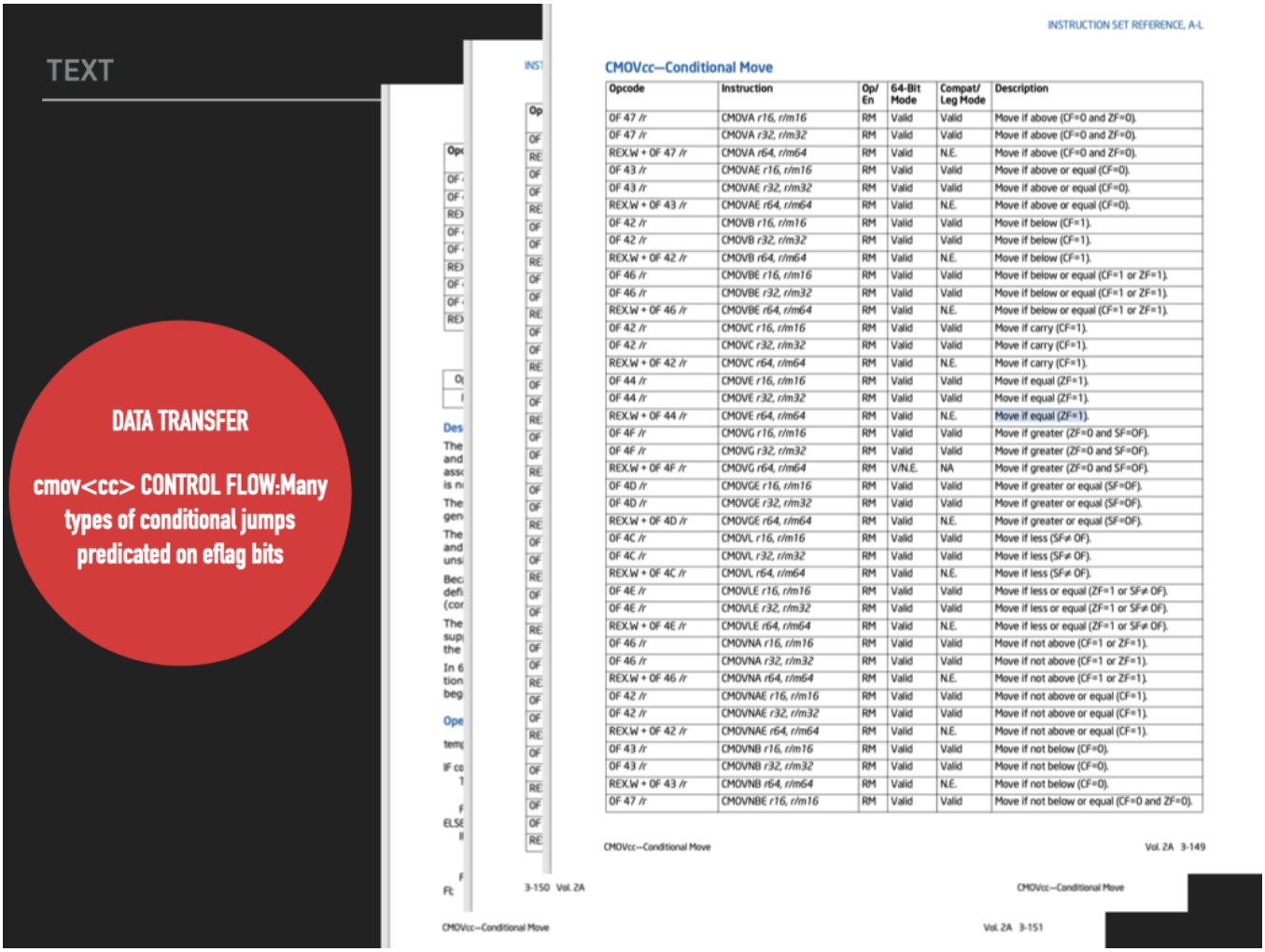
TEXT

CONTROL FLOW:

Many types of conditional jumps predicated on eflag bits

Jcc—Jump if Condition Is Met

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
77 cb	JA rel8	D	Valid	Valid	Jump short if above (CF=0 and ZF=0).
73 cb	JAE rel8	D	Valid	Valid	Jump short if above or equal (CF=0).
72 cb	JB rel8	D	Valid	Valid	Jump short if below (CF=1).
76 cb	JBE rel8	D	Valid	Valid	Jump short if below or equal (CF=1 or ZF=1).
72 cb	JC rel8	D	Valid	Valid	Jump short if carry (CF=1).
E3 cb	JCXZ rel8	D	N.E.	Valid	Jump short if CX register is 0.
E3 cb	JECXZ rel8	D	Valid	Valid	Jump short if ECX register is 0.
E3 cb	JRCXZ rel8	D	Valid	N.E.	Jump short if RCX register is 0.
74 cb	JE rel8	D	Valid	Valid	Jump short if equal (ZF=1).
7F cb	JG rel8	D	Valid	Valid	Jump short if greater (ZF=0 and SF=OF).
7D cb	JGE rel8	D	Valid	Valid	Jump short if greater or equal (SF=OF).
7C cb	JL rel8	D	Valid	Valid	Jump short if less (SF≠ OF).
7E cb	JLE rel8	D	Valid	Valid	Jump short if less or equal (ZF=1 or SF≠ OF).
76 cb	JNA rel8	D	Valid	Valid	Jump short if not above (CF=1 or ZF=1).
72 cb	JNAE rel8	D	Valid	Valid	Jump short if not above or equal (CF=1).
73 cb	JNB rel8	D	Valid	Valid	Jump short if not below (CF=0).
77 cb	JNBE rel8	D	Valid	Valid	Jump short if not below or equal (CF=0 and ZF=0).
73 cb	JNC rel8	D	Valid	Valid	Jump short if not carry (CF=0).
75 cb	JNE rel8	D	Valid	Valid	Jump short if not equal (ZF=0).
7E cb	JNG rel8	D	Valid	Valid	Jump short if not greater (ZF=1 or SF≠ OF).
7C cb	JNGE rel8	D	Valid	Valid	Jump short if not greater or equal (SF≠ OF).
7D cb	JNL rel8	D	Valid	Valid	Jump short if not less (SF=OF).
7F cb	JNLE rel8	D	Valid	Valid	Jump short if not less or equal (ZF=0 and SF=OF).
71 cb	JNO rel8	D	Valid	Valid	Jump short if not overflow (OF=0).
7B cb	JNP rel8	D	Valid	Valid	Jump short if not parity (PF=0).
79 cb	JNS rel8	D	Valid	Valid	Jump short if not sign (SF=0).
75 cb	JNZ rel8	D	Valid	Valid	Jump short if not zero (ZF=0).
70 cb	JO rel8	D	Valid	Valid	Jump short if overflow (OF=1).
7A cb	JP rel8	D	Valid	Valid	Jump short if parity (PF=1).
7A cb	JPE rel8	D	Valid	Valid	Jump short if parity even (PF=1).
7B cb	JPO rel8	D	Valid	Valid	Jump short if parity odd (PF=0).
78 cb	JS rel8	D	Valid	Valid	Jump short if sign (SF=1).
74 cb	JZ rel8	D	Valid	Valid	Jump short if zero (ZF = 1).
0F 87 cw	JA rel16	D	N.S.	Valid	Jump near if above (CF=0 and ZF=0). Not supported in 64-bit mode.
0F 87 cd	JA rel32	D	Valid	Valid	Jump near if above (CF=0 and ZF=0).
0F 83 cw	JAE rel16	D	N.S.	Valid	Jump near if above or equal (CF=0). Not supported in 64-bit mode.



10.1. `sumit.s` and `usesum.s`

10.1.1. Setup

- create a directory `mkdir sum; cd sum`
- create and write `sumit.s` and `usesumit.s` see below
- add a `Makefile` to automate assembling and linking
  - we are going run the commands by hand this time to highlight the details
- add our `setup.gdb` to make working in gdb easier
- normally you would want to track everything in git

10.1.2. Lets try and write a reusable routine

- lets assume that we have a symbol `XARRAY` that is the address of the data
- lets assume to use our routine you need to pass the length of the array
  - len in `rbx`
  - use `rdi` for loop index (`i`)
- let put the result in `rax`

Think about our objective in these terms

$$S = \sum_{i=0}^{n-1} x_i \rightarrow rax = \sum_{rdi=0}^{rbx-1} XARRAY[rdi]$$

right?

Ok remember the tricky part is realizing that it is up to us to implement the idea of an array.

- it is a data structure that we need to keep straight our head

10.1.2.1. To assemble `sumit.s` into `sumit.o`

```
as -g  sumit.s -o sumit.o
```

10.1.3. So how might we use our “fragment”

Lets create a program that defines a `_start` routine and creates the memory locations that we can control.

Lets create `usesum.s`

Lets assume that

- **will set aside enough memory for an maximum of 1000 values in where we set the `XARRAY` symbol**
- **we will allow the length actual length of data in `XARRAY` to be specified at a location marked by `XARRAY_LEN`.**
- **we will store the result in a location marked by the symbol `sum`**

We will use our code by loading our data at XARRAY, updating XARRAY_LEN, executing the code and examining the result

1. **The code should setup the memory we need**
2. **setup the registers as needed for `sumIt`**
3. **run `sumIt`**
4. **store the results at the location of `sum`**

CODE: asm - usesum.s

```
        .intel_syntax noprefix

        .section .data
        #  a place for us to store how much data is in the XARRAY
        #  initalized it to 0
XARRAY_LEN:
        .quad 0x0
        # reserve enough space for 1024 8 byte values
        #  third argument is alignment.... turns out cpu
        #  cpu is more efficient if things are located at address
        #  of a particular 'alignment (see intel manual)
        .comm XARRAY, 8*1024, 8 - each Element
        .comm sum, 8, 8          # space to store final sum
                    Element
        .section .text
        .global _start
_start:
        mov rbx, QWORD PTR [XARRAY_LEN]

        jmp sumIt
        mov QWORD PTR [sum], rax

        int3
```

10.1.3.1. To assemble `usesum.s` into `usesum.o`

```
as -g  usesum.s -o usesum.o
```

10.1.3.2. To link `usesum.o` and `sumit.o` into an executable `usesum`

```
ld -g  usesum.o sumit.o -o usesum
```

10.1.4. Lets make some data!

```
$ od -t d8 100randomnum.bin | head -10
0000000  6259685114560636175 -5631557134040358807
0000020 -4712855910177624953 -1255241704514375442
0000040  7845991343176347585 -3293899044239553636
0000060 -2978603470776467133 -5260348793470561499
0000100  5985439629054967546 -2291256575101715932
0000120  5705005805913173539 -7574407545681432134
0000140  8170410682191702331  1406609228846829472
0000160   -96234991007374263  5830597432406181851
0000200  3243745949415779362 -3237505134039398800
0000220 -3990275624606574921 -2683132816410759618
```

10.1.5. How to run `usesum` and load data with gdb

```
gdb -x setup.gdb usesum
b _start
run
# restore lets us load memory from a file
restore 10num.bin binary &XARRAY
# set the number of elements
set {long long} & XARRAY_LEN = 10
# get address of XARRAY
p & XARRAY
# now use this address to display the element we are currently summing
# eg lets assume the prior command indicated that the address of XARRAY is 0x402010
display {long long}(0x402010 + ($rdi * 8))
# display the XARRAY_LEN and sum so we can see if and when they change
display {{long long} & XARRAY_LEN, {long long} & sum}
# now we can single step our way through or continue till we hit an int3
```

You can do the same now with the `100randomnum.bin` file. Eg.`restore 100randomnum.bin & XARRAY` and `set {long long} & XARRAY_LEN = 100`

10.2. This code has a problem that we will address in the next lecture