INF113: From CPU to C

Kirill Simonov

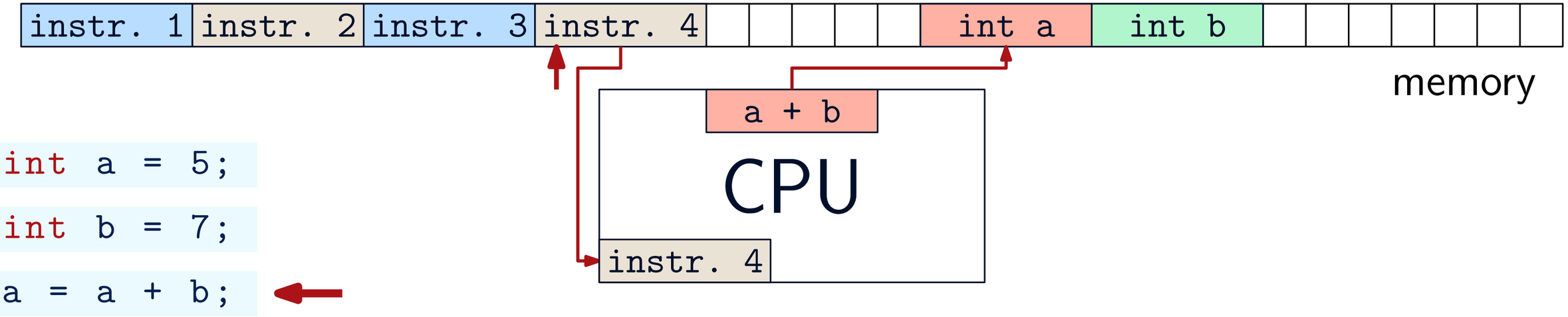
27.08.2025

Plan

- Previously: operating system stands between applications and the hardware
- We learned how to program in C a little
- **Today:** How our C programs *actually* get to run?
- We start with the CPU architecture

A coarse view

- The program uses the CPU to
  - Create variables
  - Perform arithmetic operations on variables

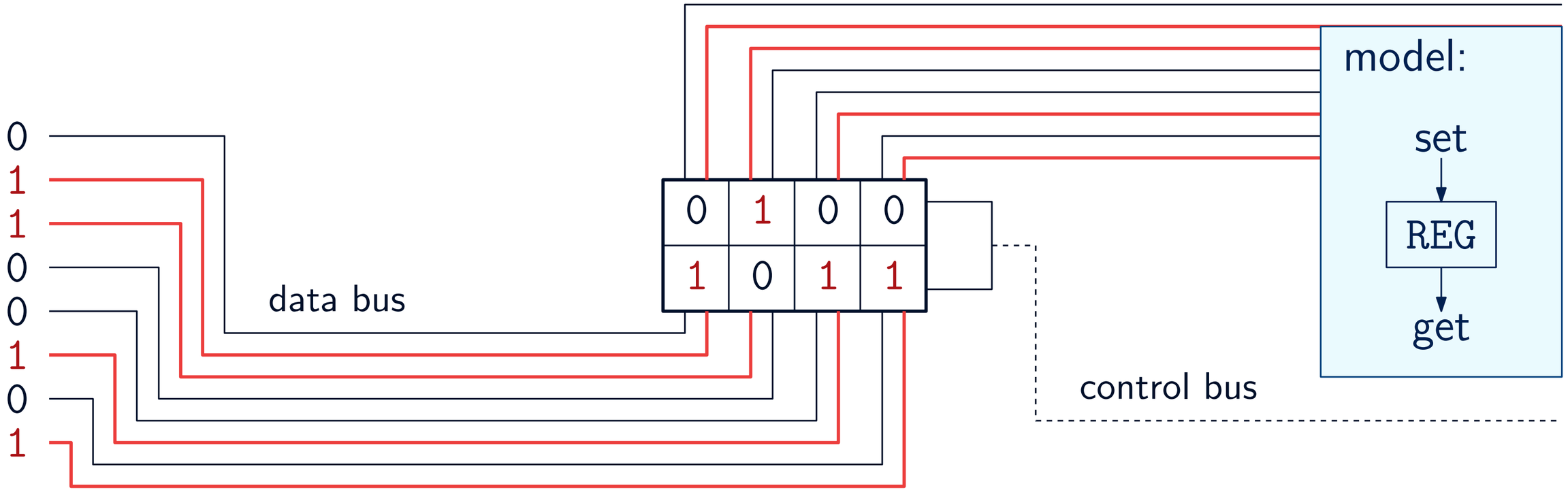


Von Neumann Model

- Both data and the program instructions lie in the same memory
- It is the CPU that reads the program instructions one by one and implements them

Registers and buses

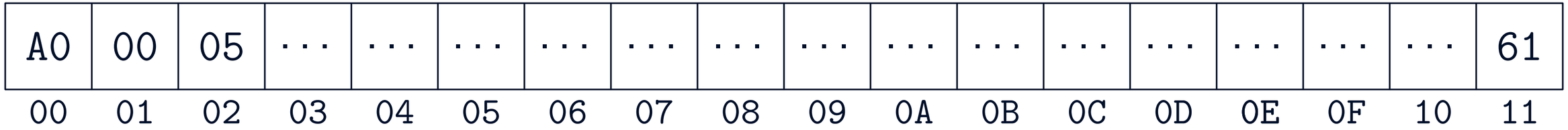
- **Register** is a special hardware memory unit, can be controlled directly



- **Bus** transfers signal/data between the units
  - Data bus lets a value to be moved into/from a register
  - Control bus sets the register to read or write

Memory

- **Memory** implements the mapping "address" $\rightarrow$ "contents"



At $\#0 \rightarrow 160$
At $\#1 \rightarrow 0$
At $\#2 \rightarrow 5$
At $\#17 \rightarrow 97$

decimal

0000 0000 $\rightarrow$ 1010 0000
0000 0001 $\rightarrow$ 0000 0000
0000 0010 $\rightarrow$ 0000 0101
0001 0001 $\rightarrow$ 0110 0001

binary

0x00 $\rightarrow$ 0xA0
0x01 $\rightarrow$ 0x00
0x02 $\rightarrow$ 0x05
0x11 $\rightarrow$ 0x61

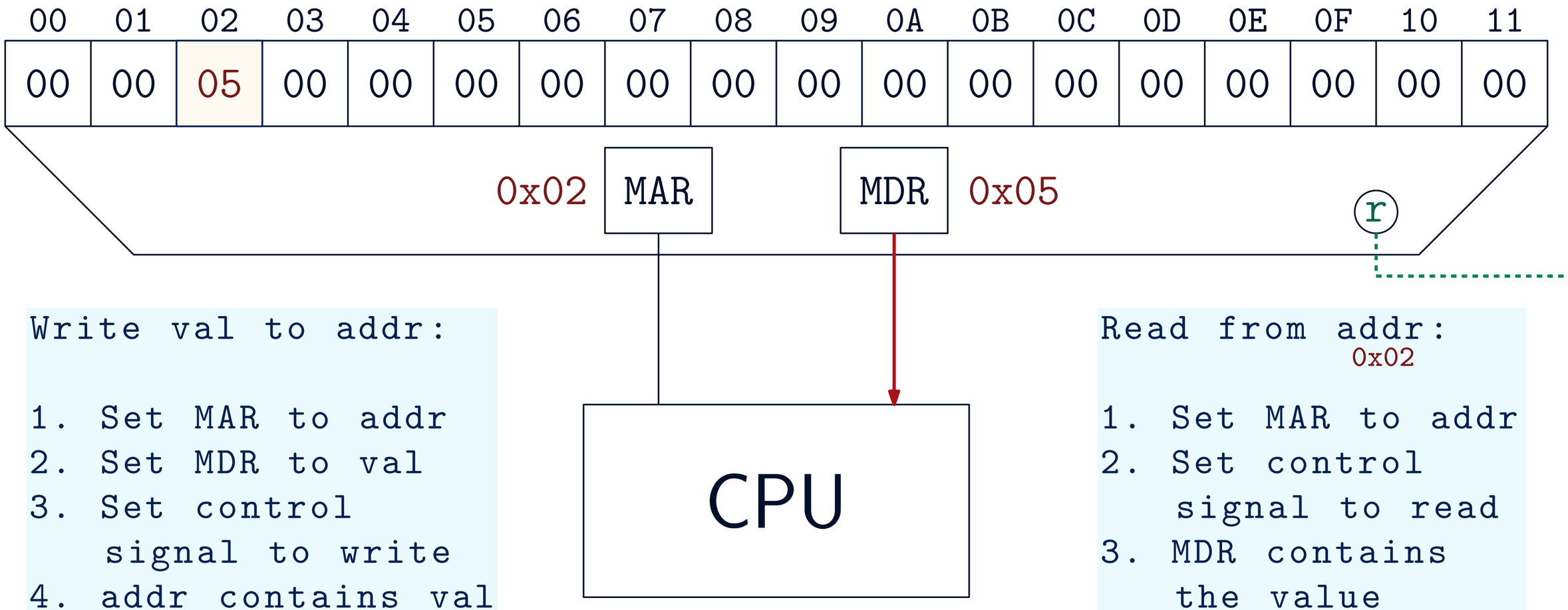
hex

- Convenient to write everything in hex
- Same contents may be interpreted in many ways
  - 0x61—number 97 or the letter 'a'
  - 0xA0—number 160 or the `MOV` instruction

Bin	Dec	Hex
0000	0	0x00
0001	1	0x01
0010	2	0x02
0011	3	0x03
0100	4	0x04
0101	5	0x05
0110	6	0x06
0111	7	0x07
1000	8	0x08
1001	9	0x09
1010	10	0x0A
1011	11	0x0B
1100	12	0x0C
1101	13	0x0D
1110	14	0x0E
1111	15	0x0F

Memory registers

- CPU interacts with memory via address register (MAR) and data register (MDR)

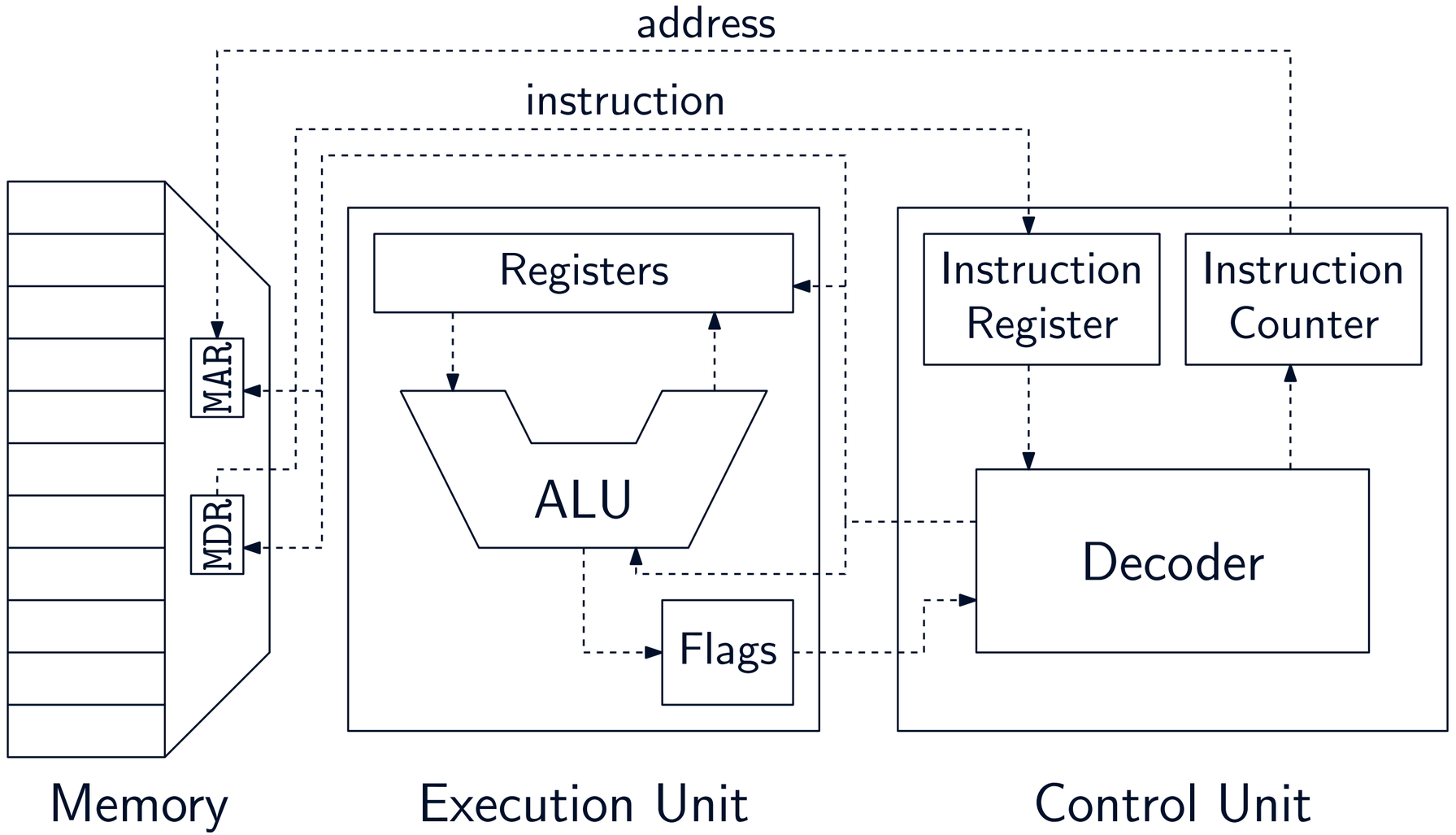



```
Write val to addr:

1. Set MAR to addr
2. Set MDR to val
3. Set control
   signal to write
4. addr contains val
```

```
Read from addr:
                0x02

1. Set MAR to addr
2. Set control
   signal to read
3. MDR contains
   the value
```

The CPU cycle

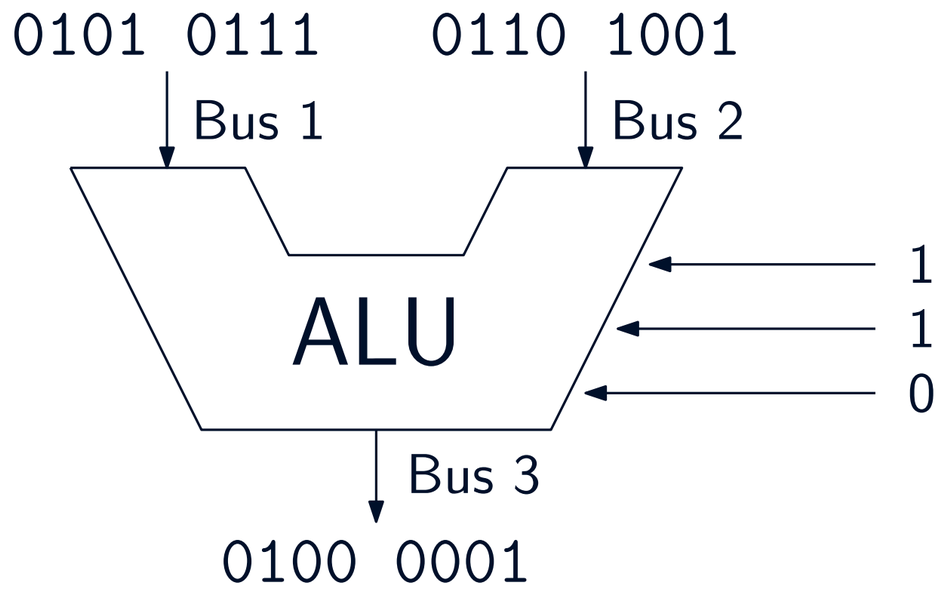
1. Set MAR to address in Instruction Counter
2. Increment IC
3. Copy MDR to Instruction Register
4. Instruction from IR is decoded
5. Control Unit sends signals based on the instruction



a.k.a. Fetch-Decode-Execute cycle

Arithmetic Logic Unit

- Computes basic arithmetic operations



F2	F1	F0	Output
0	0	0	A
0	0	1	B
0	1	0	A+1
0	1	1	B+1
1	0	0	A+B
1	0	1	A-B
1	1	0	A AND B
1	1	1	A OR B

- Control lines set the operation within the same unit

Example from "Step-by-Step Design and Simulation of a Simple CPU Architecture" by Derek C. Schuurman

https://archive.org/download/computer-organization-and-design-fourth-edition-the-hardware-software-interface-/Step-by-Step%20Design%20and%20Simulation%20of%20a%20Simple%20CPU.2445296.pdf

Instructions

- x86 architecture: ~ 1500 instructions

ADD	Add	(1) `r += r/m/imm;` (2) `m += r/imm;`	0x00...0x05, 0x80/0...0x81/0, 0x83/0
AND	Logical AND	(1) `r &= r/m/imm;` (2) `m &= r/imm;`	0x20...0x25, 0x80...0x81/4, 0x83/4
CALL	Call procedure	`push eip; eip points to the instruction directly after the call`	0x9A, 0xE8, 0xFF/2, 0xFF/3
CBW	Convert byte to word	`AX = AL ; sign extended`	0x98
CLC	Clear carry flag	`CF = 0;`	0xF8
CLD	Clear direction flag	`DF = 0;`	0xFC
CLI	Clear interrupt flag	`IF = 0;`	0xFA
CMC	Complement carry flag	`CF = !CF;`	0xF5
CMP	Compare operands	(1) `r - r/m/imm;` (2) `m - r/imm;`	0x38...0x3D, 0x80...0x81/7, 0x83/7

mnemonic

instruction code

https://en.wikipedia.org/wiki/X86_instruction_listings

Instructions: MOV

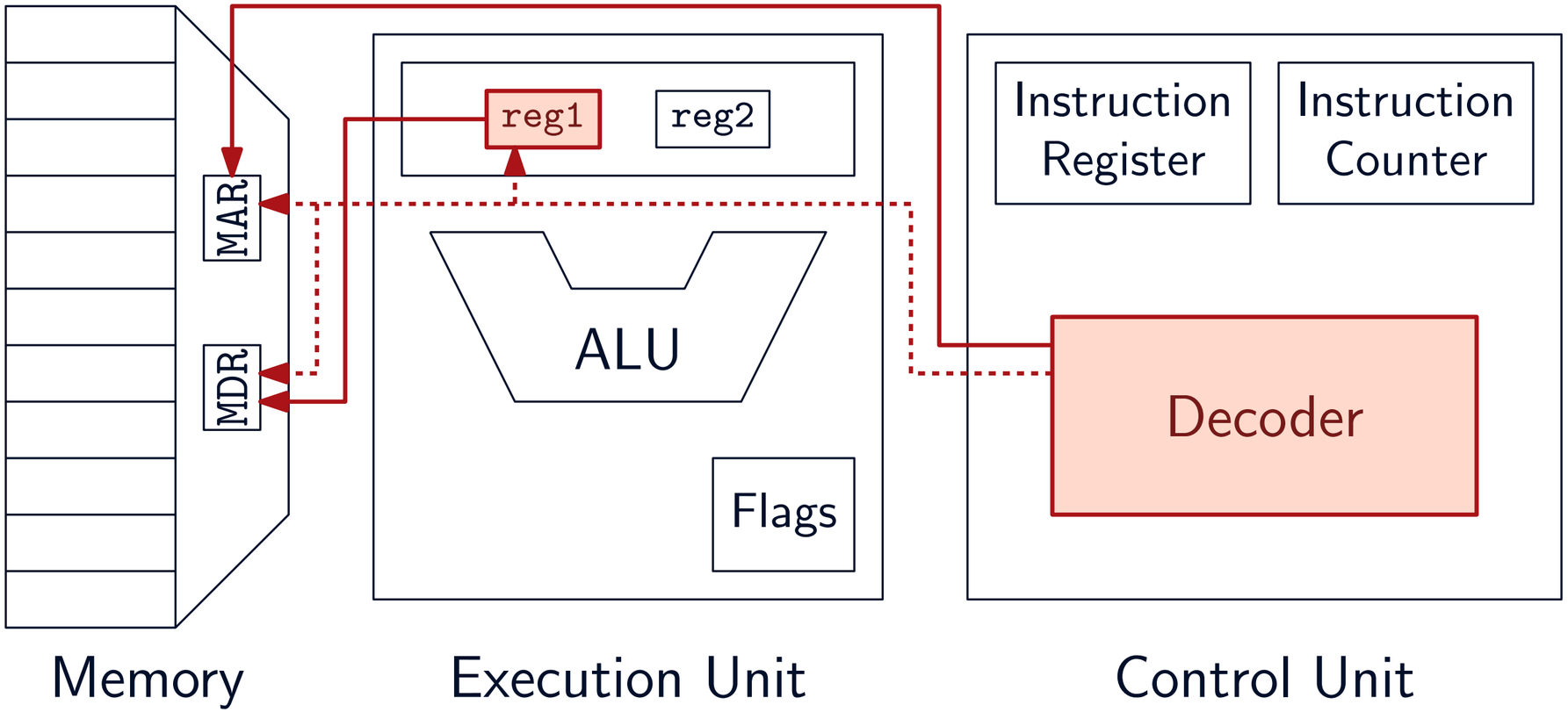
- MOV: copy source into destination

```
mov reg1, reg2
```

```
mov reg1, [memory location]
```

```
mov [memory location], reg1
```

- Cannot use two memory locations in the same instruction



Instructions: ADD

- ADD: add source to destination, store in destination

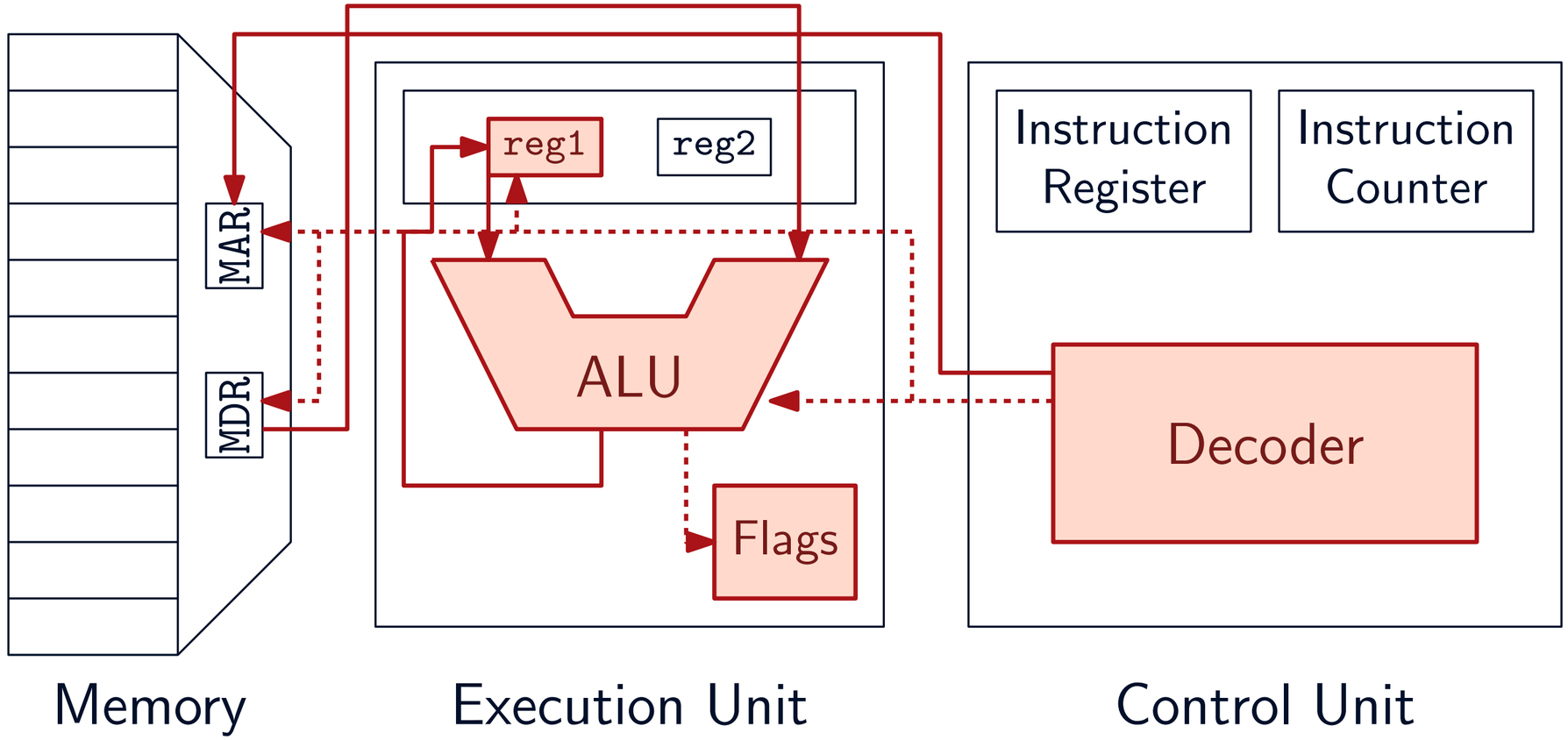
```
add reg1, reg2
```

```
add reg1, [memory location]
```

```
add [memory location], reg1
```

- Cannot use two memory locations in the same instruction

- Same principle for
  - SUB, subtraction
  - IMUL, multiplication
  - AND, bitwise and
  - ...

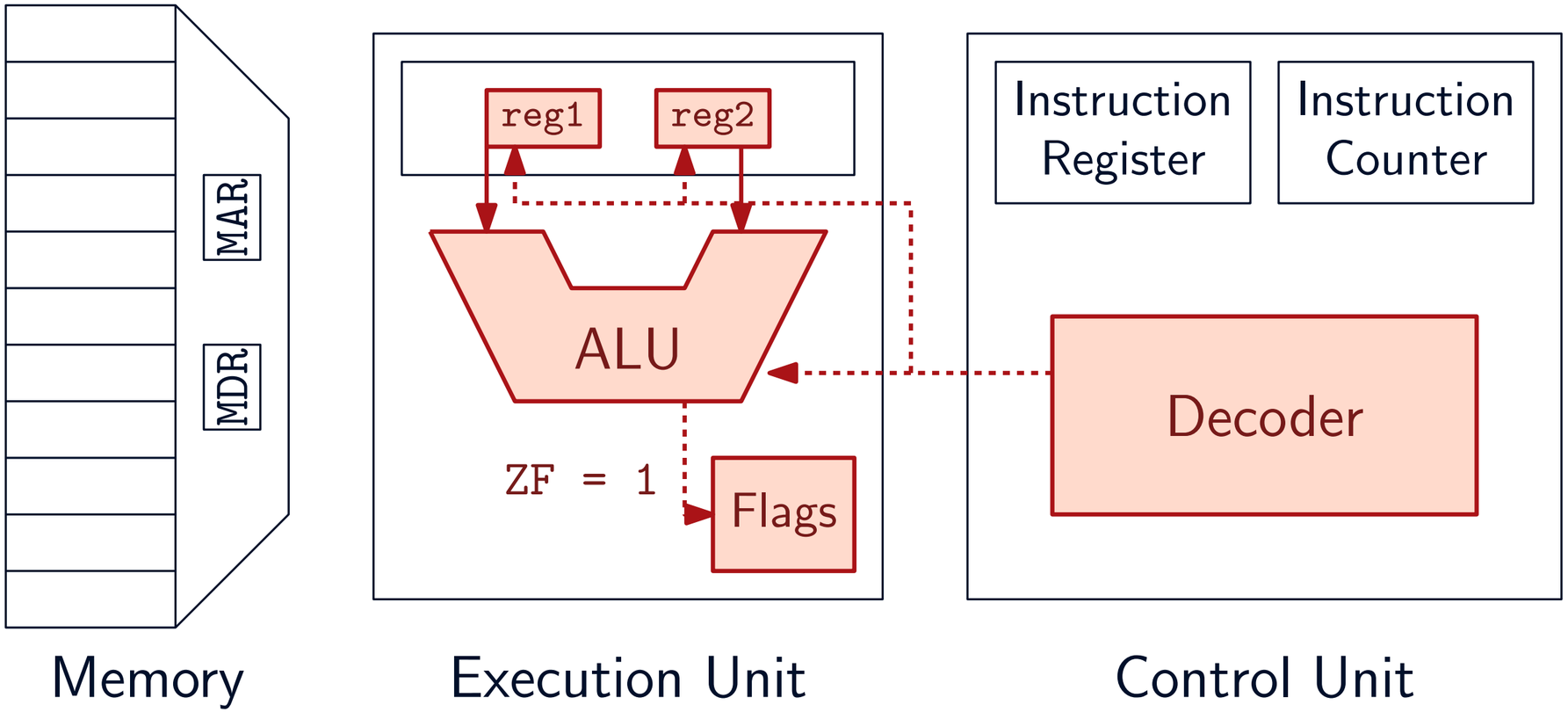


Instructions: CMP

- CMP: subtract source from destination, but without changing contents
  Only store the flags

```
cmp reg1, reg2
```

- There is also
  Negative Flag,
  Parity Flag,
  Overflow Flag,
  ...



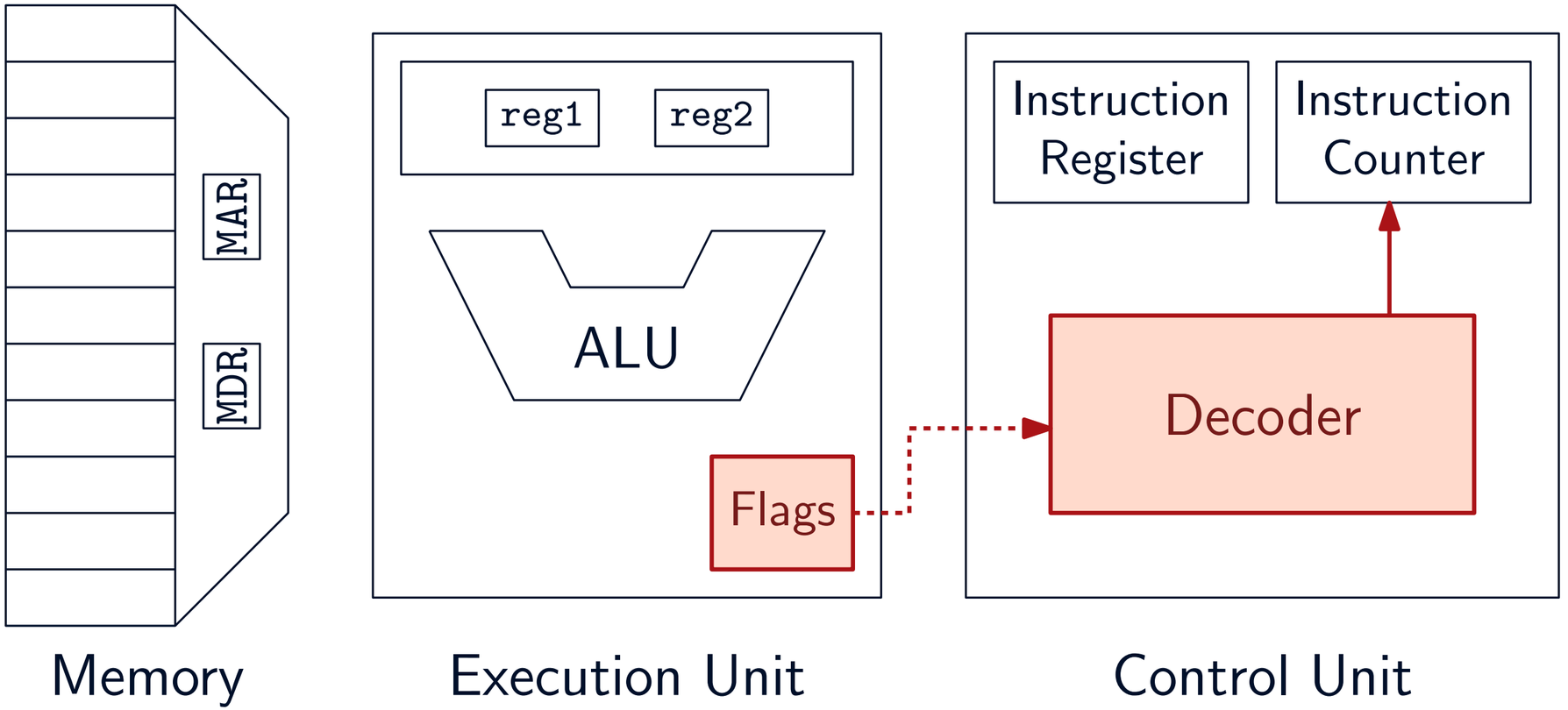
Instructions: JE

- JE: set IC to location of label if `ZF` flag is 1

```
je label
```

- Allows to model control statements: `if`, `for`, `while`

- There is also
  - JMP, unconditional
  - JNE, if not zero
  - JL, if negative
  - ...



Other instructions

- Tons of variations for MOV, JMP, arithmetics

- Operations that use fewer cycles by combining other common operations:
  - PUSH for copying a register into a stack, to simplify function calls; also PULL
  - CALL for jumping to start of the function and saving the context to the stack at the same time
  - ...

- Floating point arithmetics
  - Vectorized operations: perform several individual calucalation at the same time

- Interaction with the system
  - Send/receive system signals
  - System calls—more on Friday

Assembly language

- Human-readable version of machine instructions
  - Mnemonic instructions instead of codes
  - Label/variable names
  - Expressions for moving pointers in memory

goodbolt.org
Compiler explorer

C code

```c
int add(int a, int b) {
    return a + b;
}
```

```
gcc -S -o source.s source.c
```

```x86
add:
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-4], edi
        mov     DWORD PTR [rbp-8], esi
        mov     edx, DWORD PTR [rbp-4]
        mov     eax, DWORD PTR [rbp-8]
        add     eax, edx
        pop     rbp
        ret
```

x86 assembly

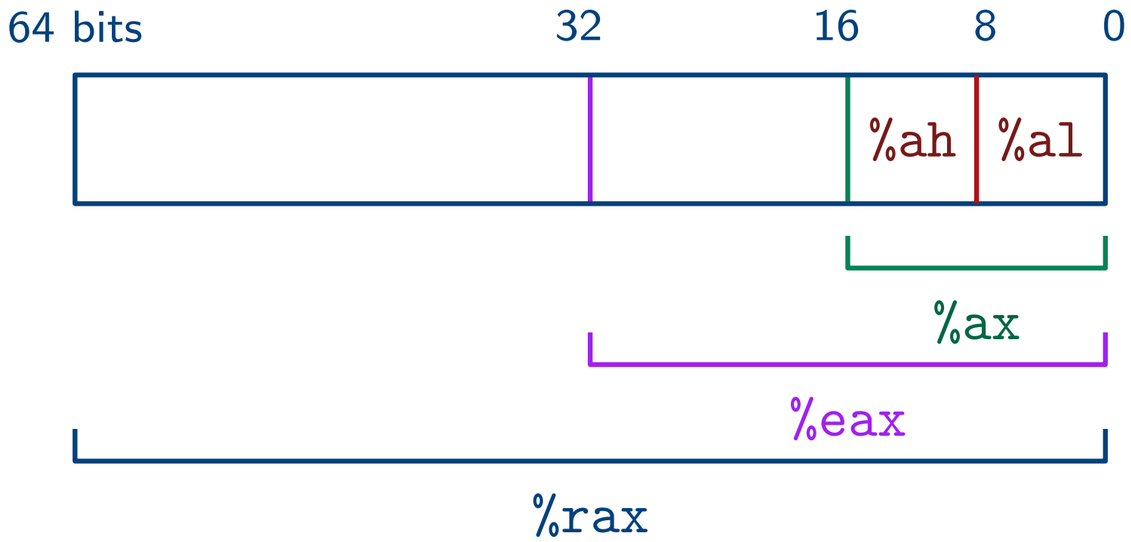
```arm
add:
        sub     sp, sp, #16
        str     w0, [sp, 12]
        str     w1, [sp, 8]
        ldr     w1, [sp, 12]
        ldr     w0, [sp, 8]
        add     w0, w1, w0
        add     sp, sp, 16
        ret
```

arm assembly

x86-64 registers

Register	Accumulator	Base	Counter	Stack Pointer	Stack Base Pointer	Destination	Source	Data
64-bit	RAX	RBX	RCX	RSP	RBP	RDI	RSI	RDX
32-bit	EAX	EBX	ECX	ESP	EBP	EDI	ESI	EDX
16-bit	AX	BX	CX	SP	BP	DI	SI	DX
8-bit	AH, AL	BH, BL	CH, CL	SPL	BPL	DIL	SIL	DH, DL

- Each register can be called in parts:



Deeper into CPU arch

- Nand2tetris Book
  https://www.nand2tetris.org/
- Simple CPU Design project
  http://www.simplecpudesign.com/
- Computers in Minecraft
  - A simple 8-bit CPU with read-only memory
    https://www.youtube.com/watch?v=ydd6l3iYOZE
  - Tutorials
    https://minecraft.fandom.com/wiki/Tutorials/Redstone_computers
- CPU Demo https://github.com/m0ne/CPUSim