LMF Laboratoire Méthodes Formelles

université PARIS-SACLAY

Introduction à la compilation

Polytech'Paris-Saclay– 4ème année –

Generation du Code

Burkhart Wolff

Plan of this Course

Plan of this Course

Plan of this Course

- A Bluffers Guide to Computer Architectures

Plan of this Course

- A Bluffers Guide to Computer Architectures
- Assembler and Machine Code

Plan of this Course

- A Bluffers Guide to Computer Architectures
- Assembler and Machine Code
- Basic Code Generation

Plan of this Course

- A Bluffers Guide to Computer Architectures
- Assembler and Machine Code
- Basic Code Generation
- Optimizations

(Standard)
Computer Architectures

Background: Computer Architecture

Background: Computer Architecture

- Basics: vonNeumann Architecture

Background: Computer Architecture

- Basics: vonNeumann Architecture
    - Modern Architectures

Background: Computer Architecture

- Basics: vonNeumann Architecture
  - Modern Architectures
  - X86, ARM 8, Risc V

Background: Computer Architecture

- Basics: vonNeumann Architecture
  - Modern Architectures
  - X86, ARM 8, Risc V
  - virtual architectures LLVM, JVM, …

Background: Computer Architecture

- Basics: vonNeumann Architecture
  - Modern Architectures
  - X86, ARM 8, Risc V
  - virtual architectures LLVM, JVM, ...
  - Processor Virtualisation

Background: Computer Architecture

- Basics: vonNeumann Architecture
  - Modern Architectures
  - X86, ARM 8, Risc V
  - virtual architectures LLVM, JVM, ...
  - Processor Virtualisation
- Basic Instruction Sets Architectures (ISA)

Background: Computer Architecture

- Basics: vonNeumann Architecture
  - Modern Architectures
  - X86, ARM 8, Risc V
  - virtual architectures LLVM, JVM, ...
  - Processor Virtualisation
- Basic Instruction Sets Architectures (ISA)

von Neumann Architecture

von Neumann Architecture

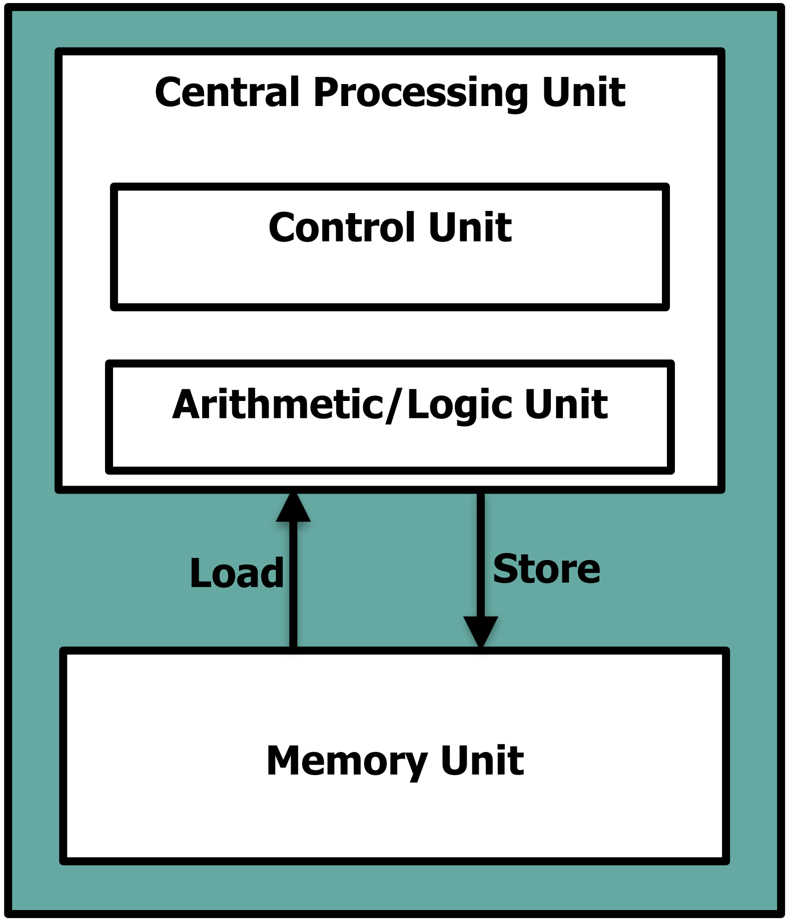
- Basic Model does back to a Technical Report by John von Neumann 1946

von Neumann Architecture

- Basic Model does back to a Technical Report by John von Neumann 1946

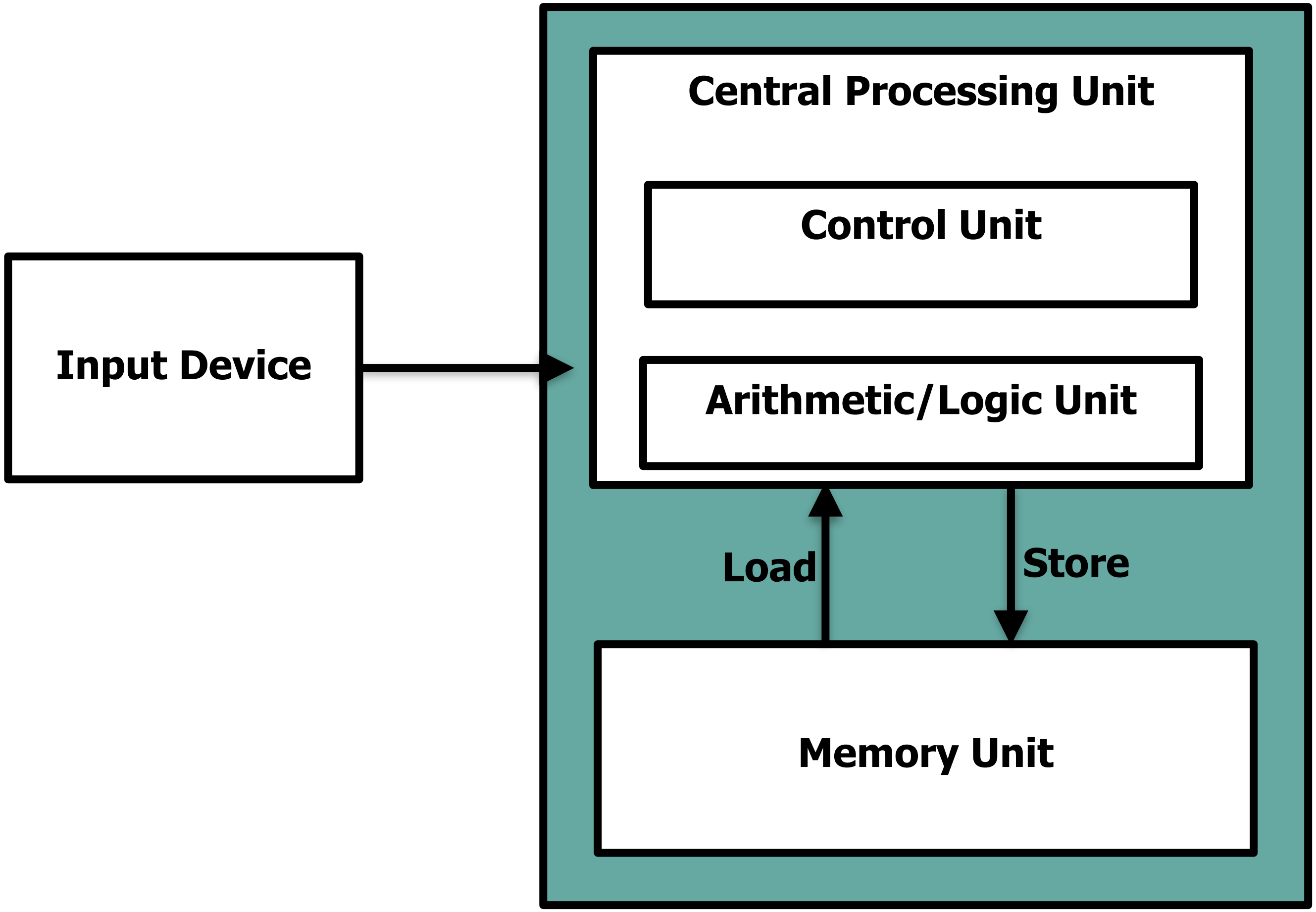
von Neumann Architecture

- Basic Model does back to a Technical Report by John von Neumann 1946



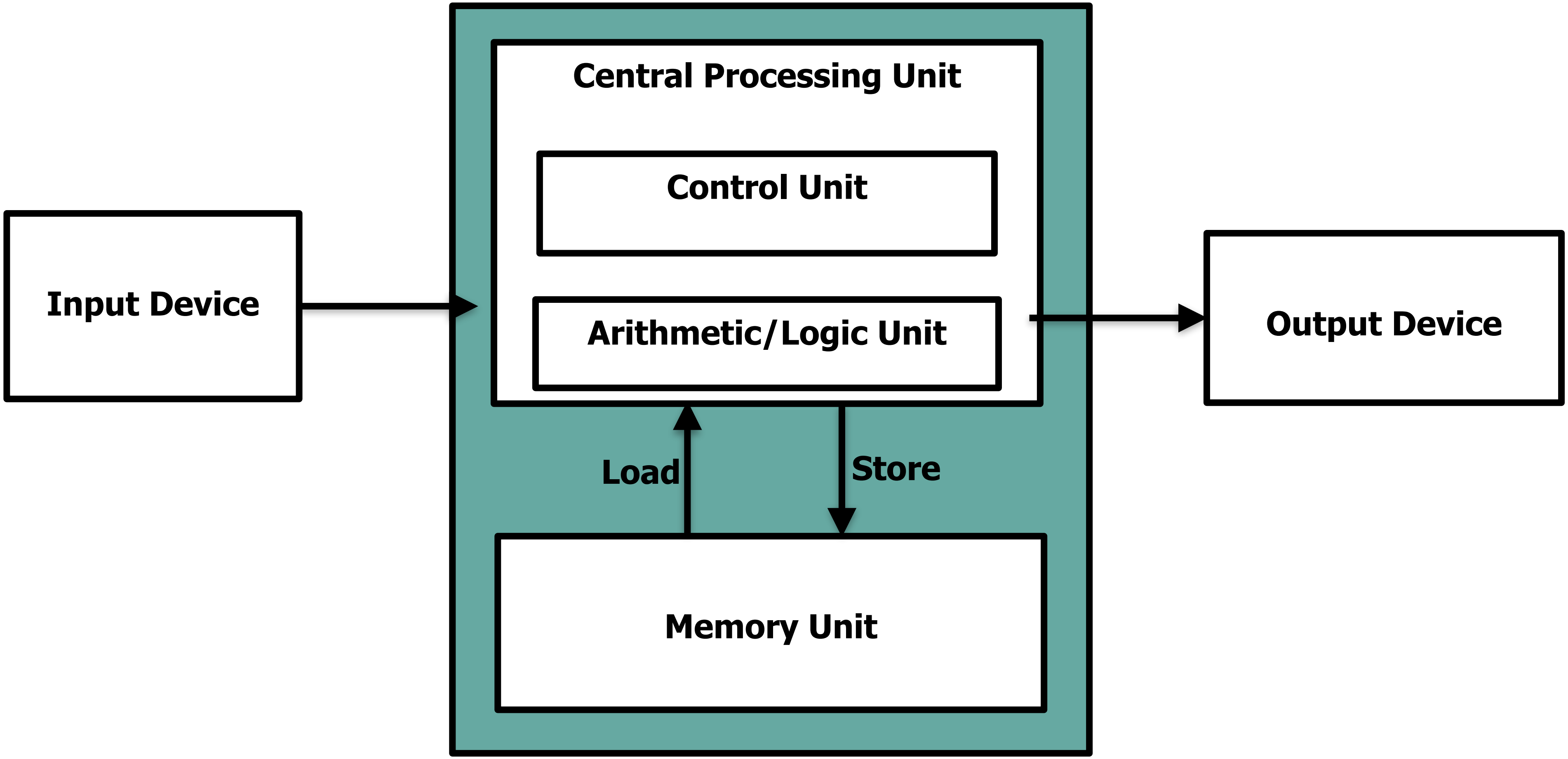
von Neumann Architecture

- Basic Model does back to a Technical Report by John von Neumann 1946



von Neumann Architecture

- Basic Model does back to a Technical Report by John von Neumann 1946



von Neumann Architecture

- Basic Model does back to a Technical Report by John von Neumann 1946

von Neumann Architecture

- Basic Model does back to a Technical Report by John von Neumann 1946

  - CPU, ALU,
  - words & adresses,
    for data and programs
  - slow memory, fast registers,
    i.e. load and store ops
  - 2s-complements for numbers,

von Neumann Architecture

von Neumann Architecture

- A more recent architecture model:

von Neumann Architecture

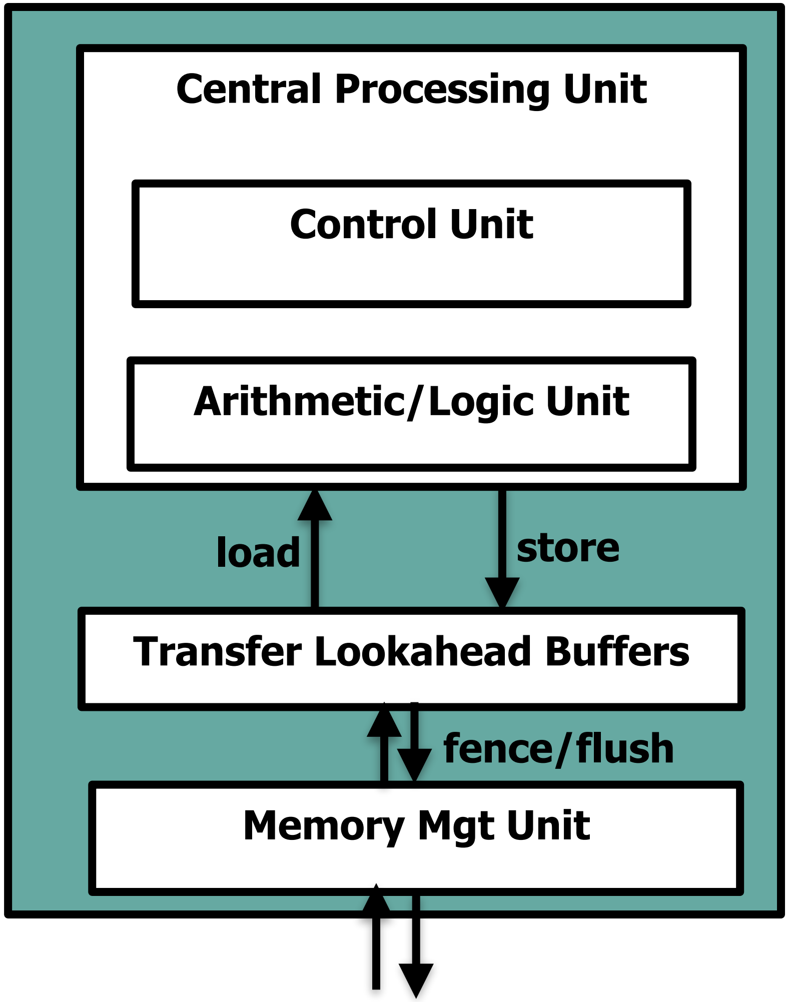
- A more recent architecture model:

von Neumann Architecture

- A more recent architecture model:

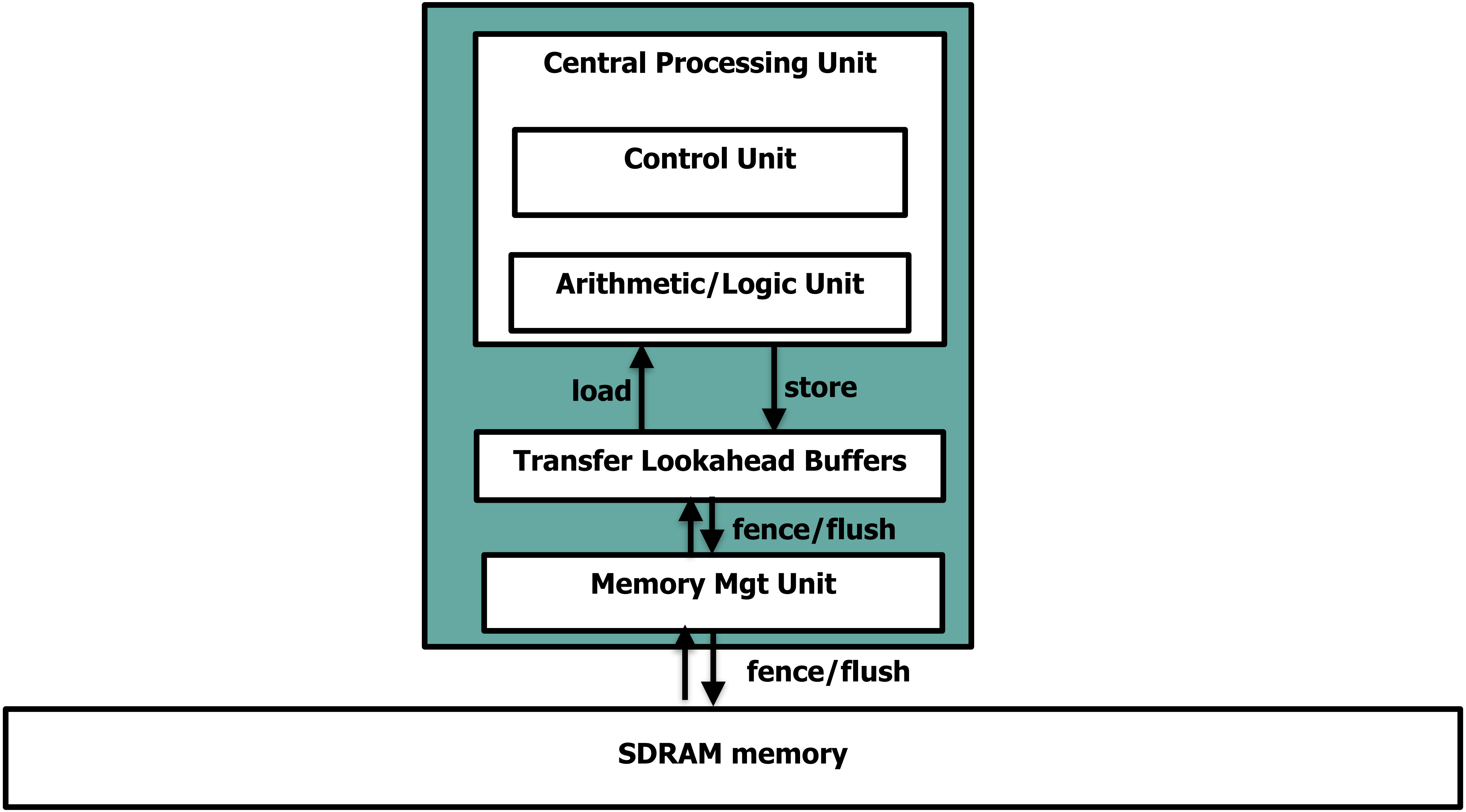
von Neumann Architecture

- A more recent architecture model:



von Neumann Architecture

- A more recent architecture model:



Central Processing Unit

Control Unit

Arithmetic/Logic Unit

load

store

Transfer Lookahead Buffers

fence/flush

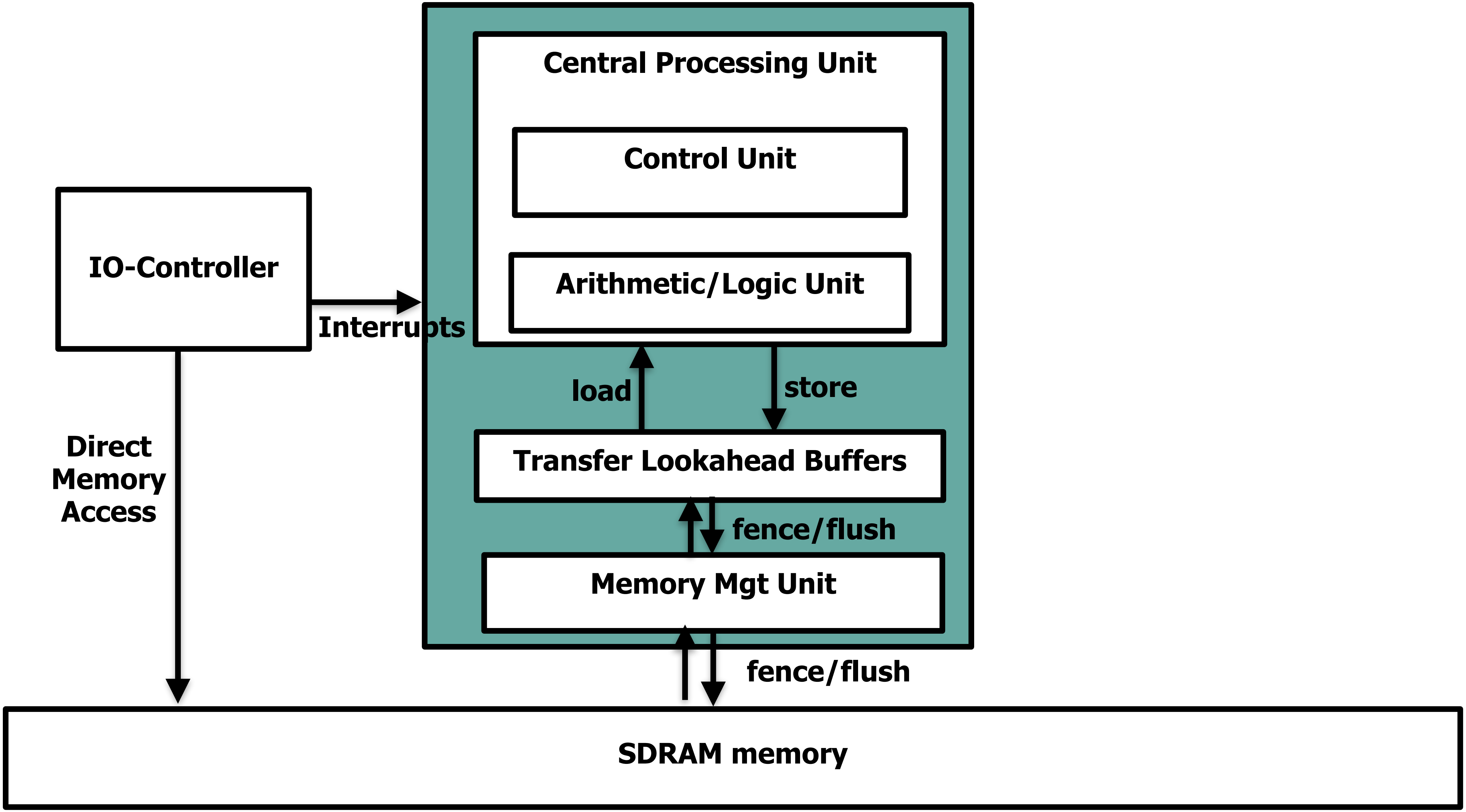
Memory Mgt Unit

fence/flush

SDRAM memory

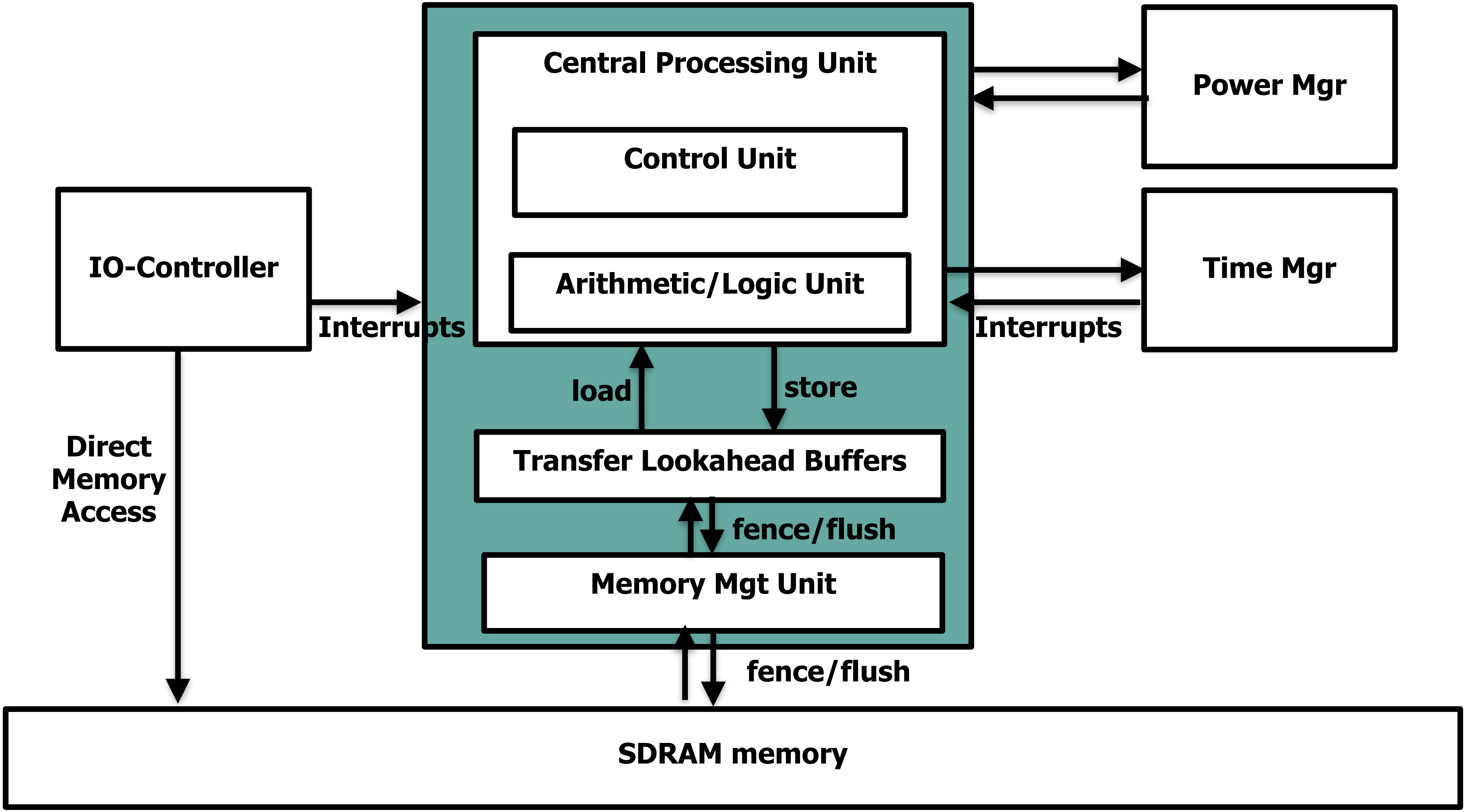
von Neumann Architecture

- A more recent architecture model:



von Neumann Architecture

- A more recent architecture model:



von Neumann Architecture

von Neumann Architecture

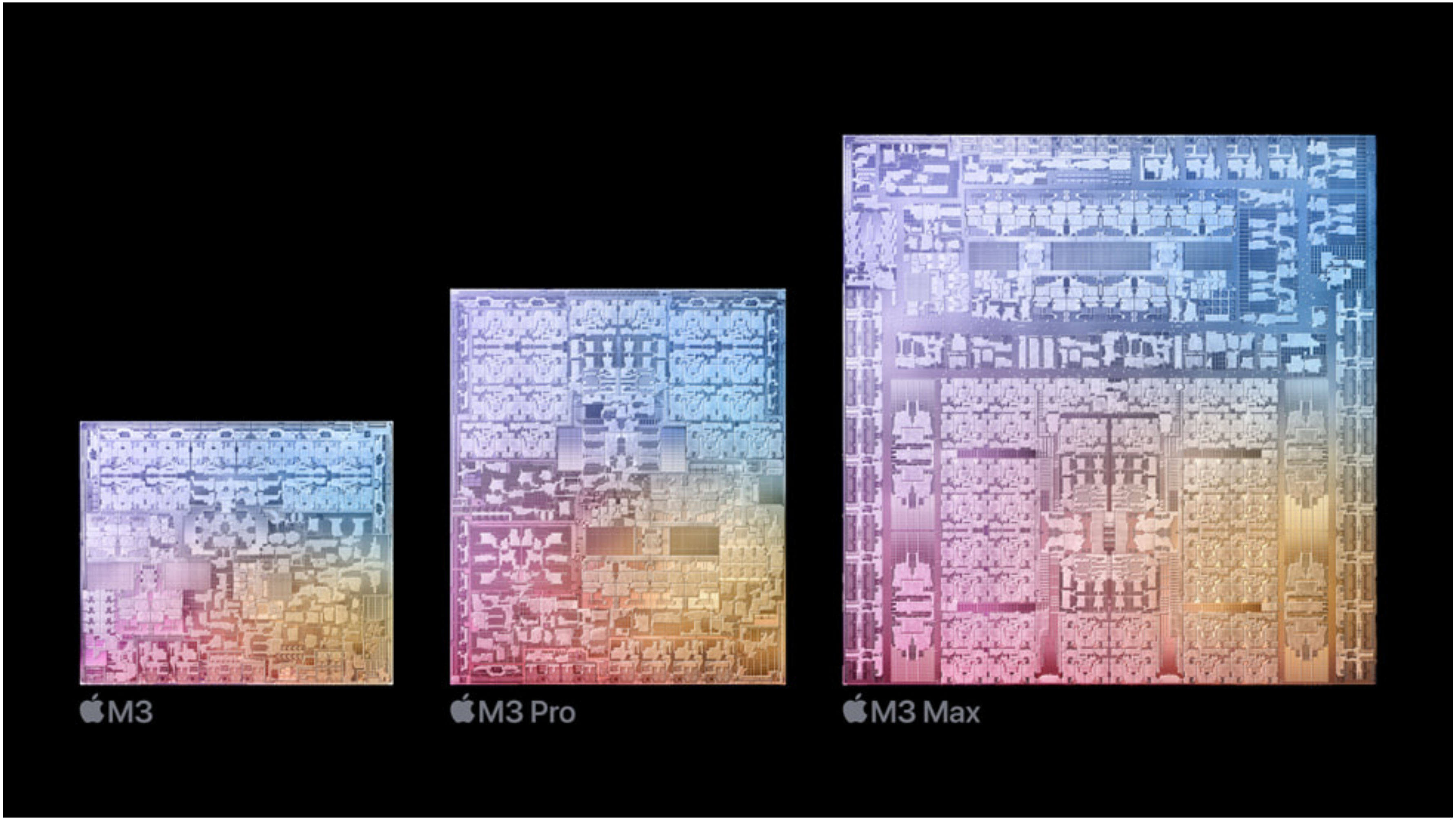
- Current ARM variants (MX, © Apple):

von Neumann Architecture

- Current ARM variants (MX, ©Apple):

von Neumann Architecture

- Current ARM variants (MX, ©Apple):



von Neumann Architecture

- Current ARM variants (MX, © Apple):
  - multi-cores basically the same
  - but graphical processing units,
  - … neural engines,
  - … and lots of stuff.

Compilation to Assembly Languages (Reminder)

Source Code to Assembly Code

Source code fib.c

```
int64_t fib(int64_t n) {
  if (n < 2) return n;
  return (fib(n-1) + fib(n-2));
}
```

```
$ clang -O3 fib.c -S
```

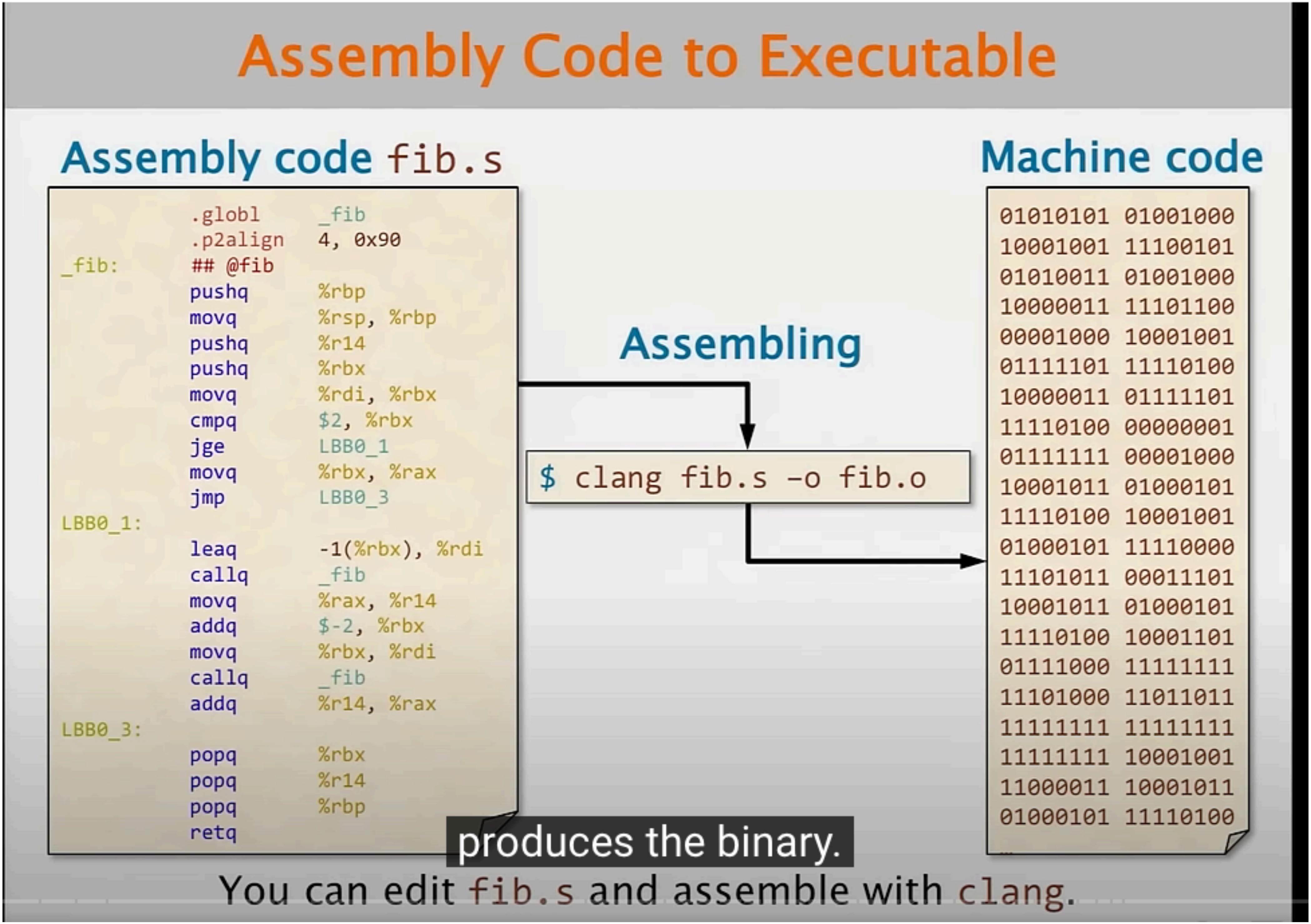
Assembly language provides a convenient symbolic representation of machine code.

Assembly code fib.s

```
            .globl      _fib
            .p2align    4, 0x90
_fib:       ## @fib
            pushq       %rbp
            movq        %rsp, %rbp
            pushq       %r14
            pushq       %rbx
            movq        %rdi, %rbx
            cmpq        $2, %rbx
            jge         LBB0_1
            movq        %rbx, %rax
            jmp         LBB0_3
LBB0_1:
            leaq        -1(%rbx), %rdi
            callq       _fib
            movq        %rax, %r14
            addq        $-2, %rbx
            movq        %rbx, %rdi
            callq       _fib
            addq        %r14, %rax
LBB0_3:
            popq        %rbx
            popq        %r14
            popq        %rbp
            retq
```

See http://sourceware.org/binutils/docs/as/index.html.

The next stage is the source code to assembly code.



Assembly Code to Executable

Assembly code fib.s

```
            .globl      _fib
            .p2align    4, 0x90
_fib:       ## @fib
            pushq       %rbp
            movq        %rsp, %rbp
            pushq       %r14
            pushq       %rbx
            movq        %rdi, %rbx
            cmpq        $2, %rbx
            jge         LBB0_1
            movq        %rbx, %rax
            jmp         LBB0_3
LBB0_1:
            leaq        -1(%rbx), %rdi
            callq       _fib
            movq        %rax, %r14
            addq        $-2, %rbx
            movq        %rbx, %rdi
            callq       _fib
            addq        %r14, %rax
LBB0_3:
            popq        %rbx
            popq        %r14
            popq        %rbp
            retq
```

Assembling

```
$ clang fib.s -o fib.o
```

Machine code

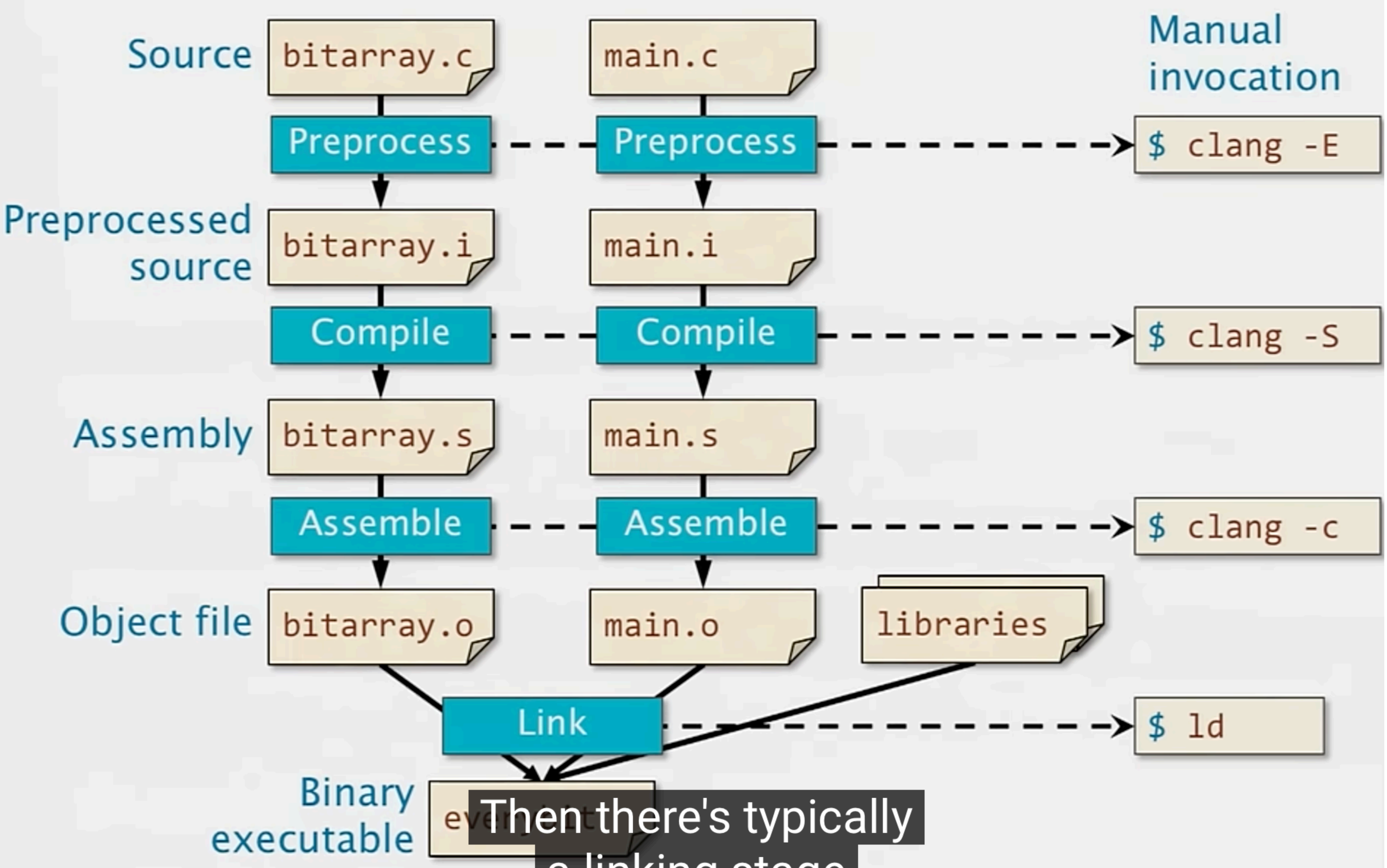
```
01010101 01001000
10001001 11100101
01010011 01001000
10000011 11101100
00001000 10001001
01111101 11110100
10000011 01111101
11110100 00000001
01111111 00001000
10001011 01000101
11110100 10001001
01000101 11110000
11101011 00011101
10001011 01000101
11110100 10001101
01111000 11111111
11101000 11011011
11111111 11111111
11111111 10001001
11000011 10001011
01000101 11110100
...
```

produces the binary.

You can edit fib.s and assemble with clang.

En réalité c'est encore trop simple … (clang LLVM)

The Four Stages of Compilation

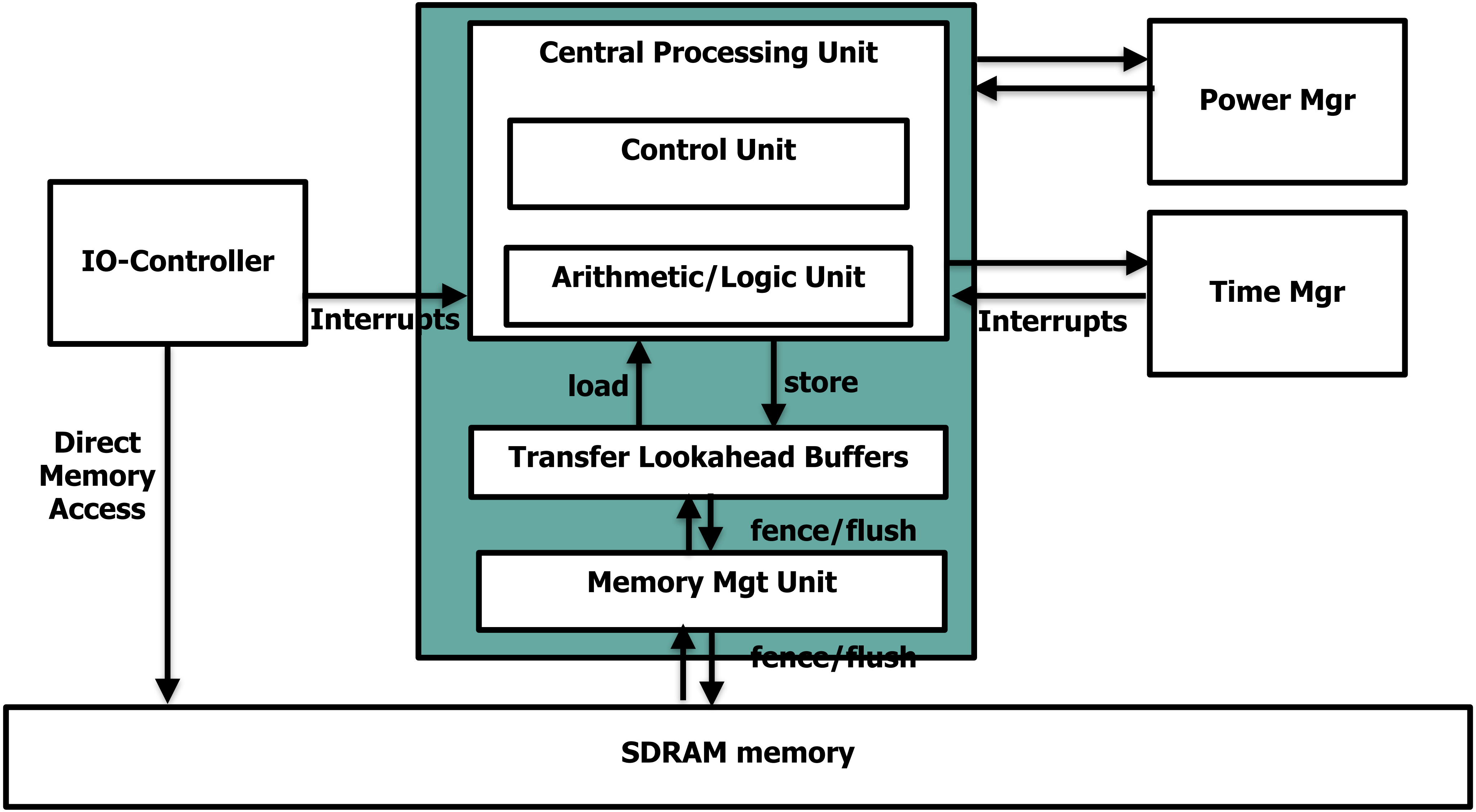


Then there's typically a linking stage

The Core:
Instruction Set Architectures
(ISA)

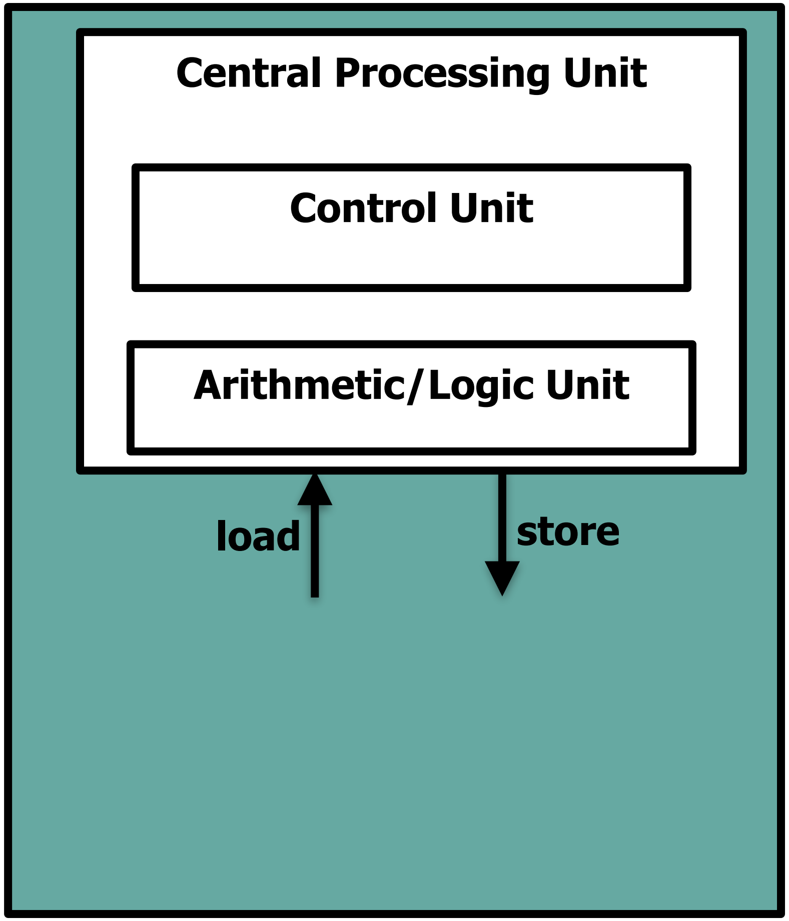
The Core of the Architecture

- Fortunately, a lot is largely irrelevant (except for embedded system developers)



The Core of the Architecture

- Fortunately, a lot is largely irrelevant (except for embedded system developers)



The Core : Computer Architecture

The Core : Computer Architecture

- Instruction Sets Architectures (ISA ’ s) are reflected directly in assembly languages

The Core : Computer Architecture

- Instruction Sets Architectures (ISA ’ s) are reflected directly in assembly languages
- There are typically different processors “implementing” a particular ISA (- family)

The Core : Computer Architecture

- Instruction Sets Architectures (ISA ' s) are reflected directly in assembly languages
- There are typically different processors "implementing" a particular ISA (- family)
- Nowadays, we see 3 major ISA families:

The Core : Computer Architecture

- Instruction Sets Architectures (ISA ' s) are reflected directly in assembly languages
- There are typically different processors "implementing" a particular ISA (- family)
- Nowadays, we see 3 major ISA families:
  - X86 (Intel and AMD processors)

The Core : Computer Architecture

- Instruction Sets Architectures (ISA ’ s) are reflected directly in assembly languages
- There are typically different processors “implementing” a particular ISA (- family)
- Nowadays, we see 3 major ISA families:
  - X86 (Intel and AMD processors)
  - ARM (lots of vendors, e.g. Apple)
  - RISC-V (open source, getting traction ...)
- ... plus virtual machine ISA’s like LLVM or JVM

An Example ISA: ARM Cortex M

An Example ISA: ARM Cortex M

- What constitutes an ISA ?

An Example ISA: ARM Cortex M

- What constitutes an ISA ?
  - where the processor stores or obtains information

An Example ISA: ARM Cortex M

- What constitutes an ISA ?
  - where the processor stores or obtains information
    - registers

An Example ISA: ARM Cortex M

- What constitutes an ISA ?
    - where the processor stores or obtains information
        - registers
        - memory

An Example ISA: ARM Cortex M

- What constitutes an ISA ?
    - where the processor stores or obtains information
        - registers
        - memory
        - input/output devices

An Example ISA: ARM Cortex M

- What constitutes an ISA ?
  - where the processor stores or obtains information
    - registers
    - memory
    - input/output devices
  - how the processor manipulates data

An Example ISA: ARM Cortex M

- What constitutes an ISA ?
  - where the processor stores or obtains information
    - registers
    - memory
    - input/output devices
  - how the processor manipulates data
    - assembly language instructions

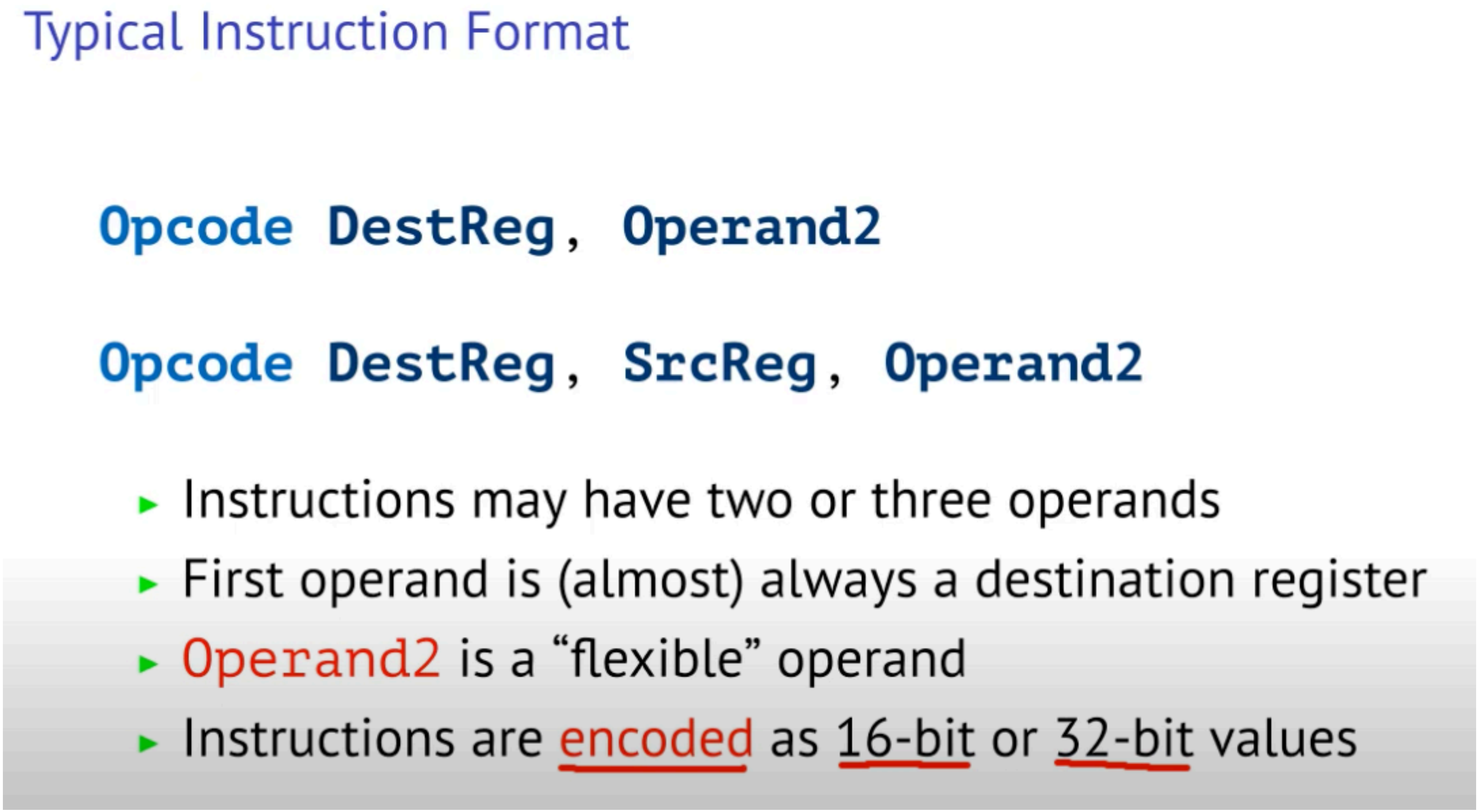
An Example ISA: ARM Cortex M

- What constitutes an ISA ?
  - where the processor stores or obtains information
    - registers
    - memory
    - input/output devices
  - how the processor manipulates data
    - assembly language instructions
    - processor hardware actions

An Example ISA: ARM Cortex M

- What constitutes an ISA ?
    - where the processor stores or obtains information
        - registers
        - memory
        - input/output devices
    - how the processor manipulates data
        - assembly language instructions
        - processor hardware actions

An Example ISA: ARM Cortex M



Typical Instruction Format

```
Opcode DestReg, Operand2

Opcode DestReg, SrcReg, Operand2
```

- Instructions may have two or three operands
- First operand is (almost) always a destination register
- Operand2 is a “flexible” operand
- Instructions are encoded as 16-bit or 32-bit values

An Example ISA: ARM Cortex 7

An Example ISA: ARM Cortex 7

- Sources:

An Example ISA: ARM Cortex 7

- Sources:
  - The ARM Cortex Manual:

    https://developer.arm.com/documentation/dui0552/a/the-cortex-m3-processor/programmers-model/core-registers?lang=en

An Example ISA: ARM Cortex 7

- Sources:
  - The ARM Cortex Manual:

    https://developer.arm.com/documentation/dui0552/a/the-cortex-m3-processor/programmers-model/core-registers?lang=en

An Example ISA: ARM Cortex 7

- Sources:
  - The ARM Cortex Manual:

    https://developer.arm.com/documentation/dui0552/a/the-cortex-m3-processor/programmers-model/core-registers?lang=en

  - A youtube manual by “JoeTheProfessor”:

An Example ISA: ARM Cortex 7

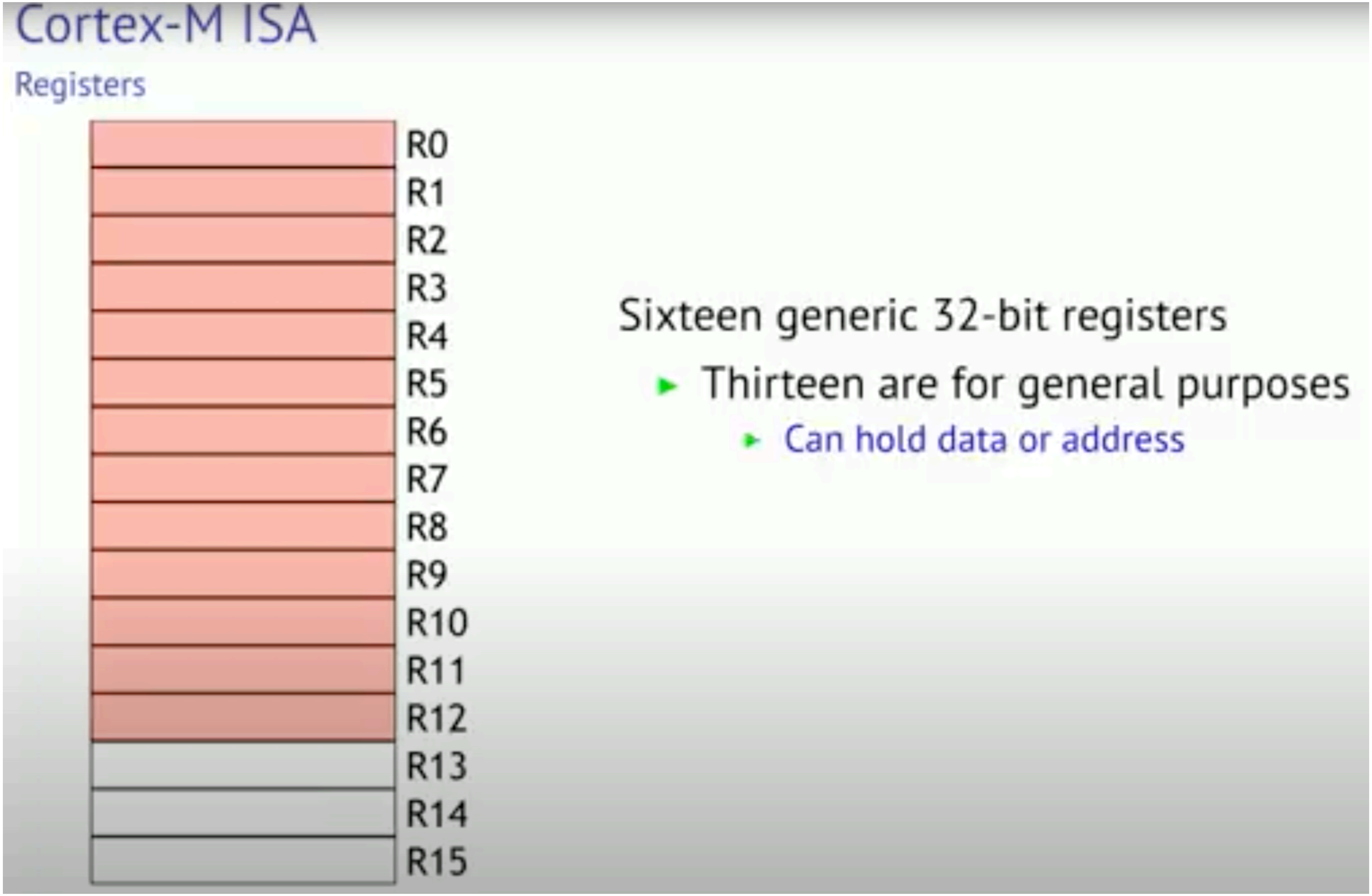
- Sources:
  - The ARM Cortex Manual:

    https://developer.arm.com/documentation/dui0552/a/the-cortex-m3-processor/programmers-model/core-registers?lang=en

  - A youtube manual by “JoeTheProfessor”:

    Tutorial Cortex M7 https://www.youtube.com/watch?v=JmpQ79h_0eA&list=PL3hGN82ZNBxi111hEdE0VhGVP_jMhN9cO&index=5

An Example ISA: ARM Cortex 7



Cortex-M ISA

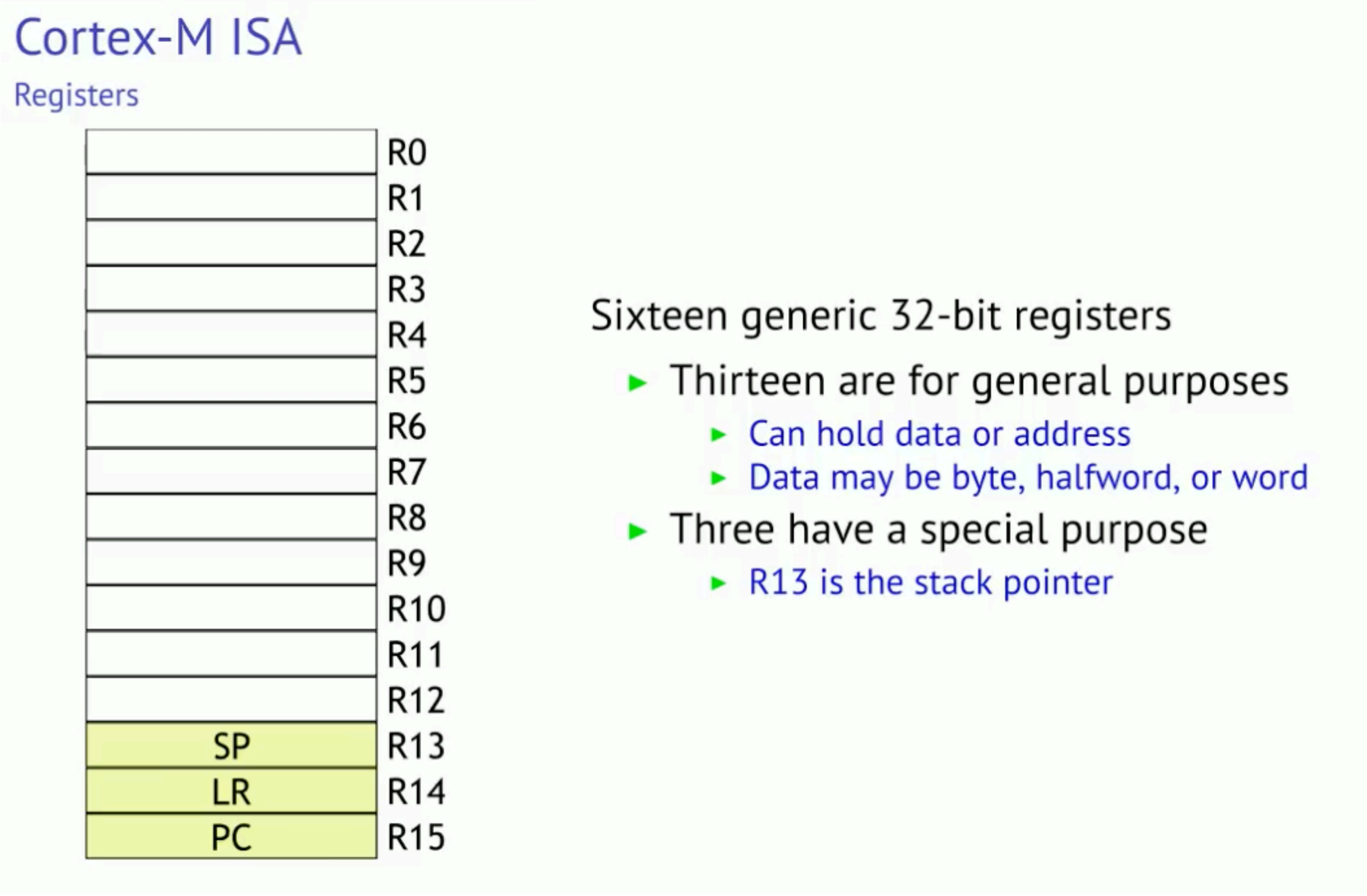
Registers

R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13
R14
R15

Sixteen generic 32-bit registers

- Thirteen are for general purposes
  - Can hold data or address

An Example ISA: ARM Cortex 7



Cortex-M ISA

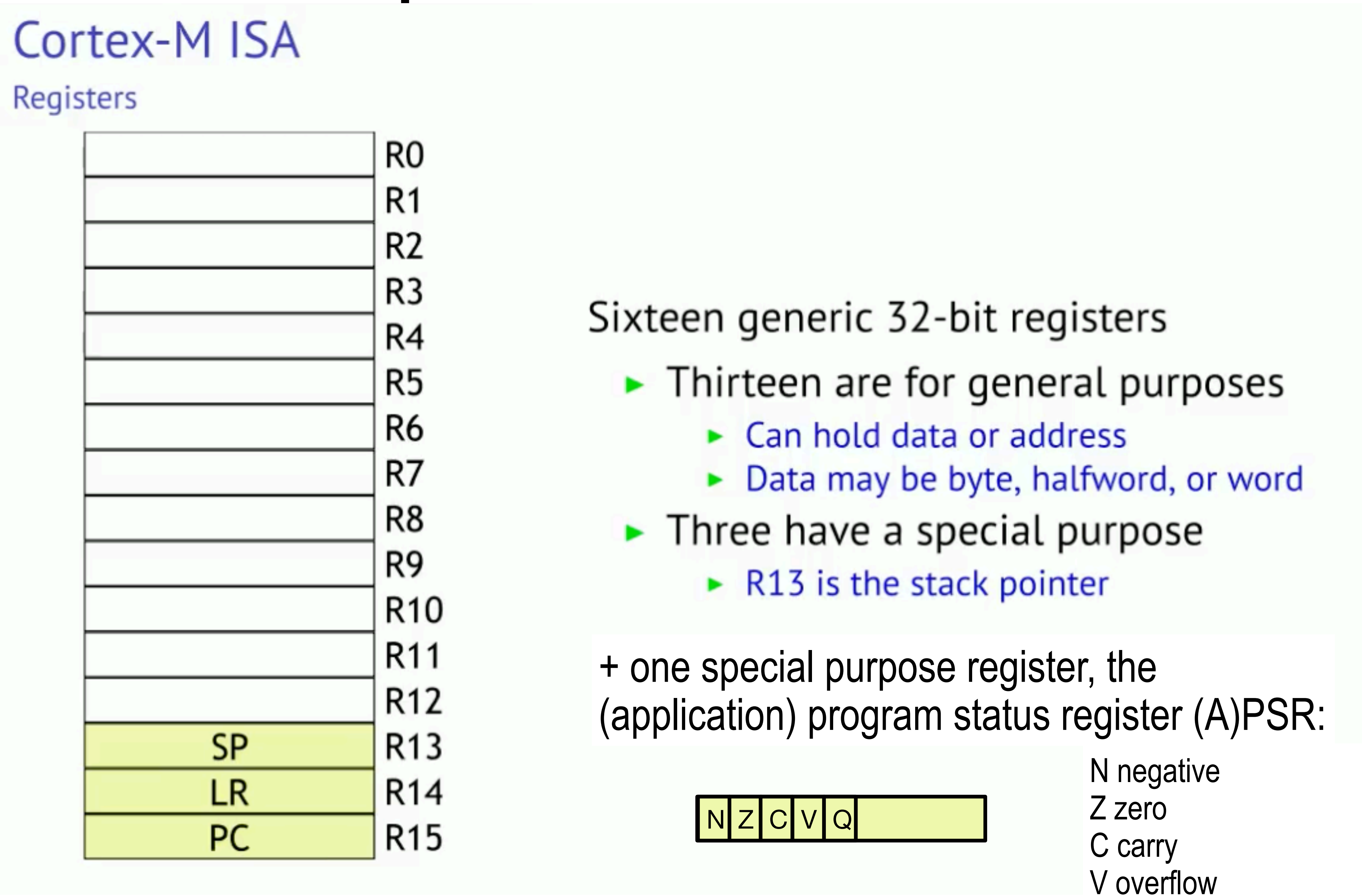
Registers

	R0
	R1
	R2
	R3
	R4
	R5
	R6
	R7
	R8
	R9
	R10
	R11
	R12
SP	R13
LR	R14
PC	R15

Sixteen generic 32-bit registers

- Thirteen are for general purposes
  - Can hold data or address
  - Data may be byte, halfword, or word
- Three have a special purpose
  - R13 is the stack pointer

An Example ISA: ARM Cortex 7



Cortex-M ISA

Registers

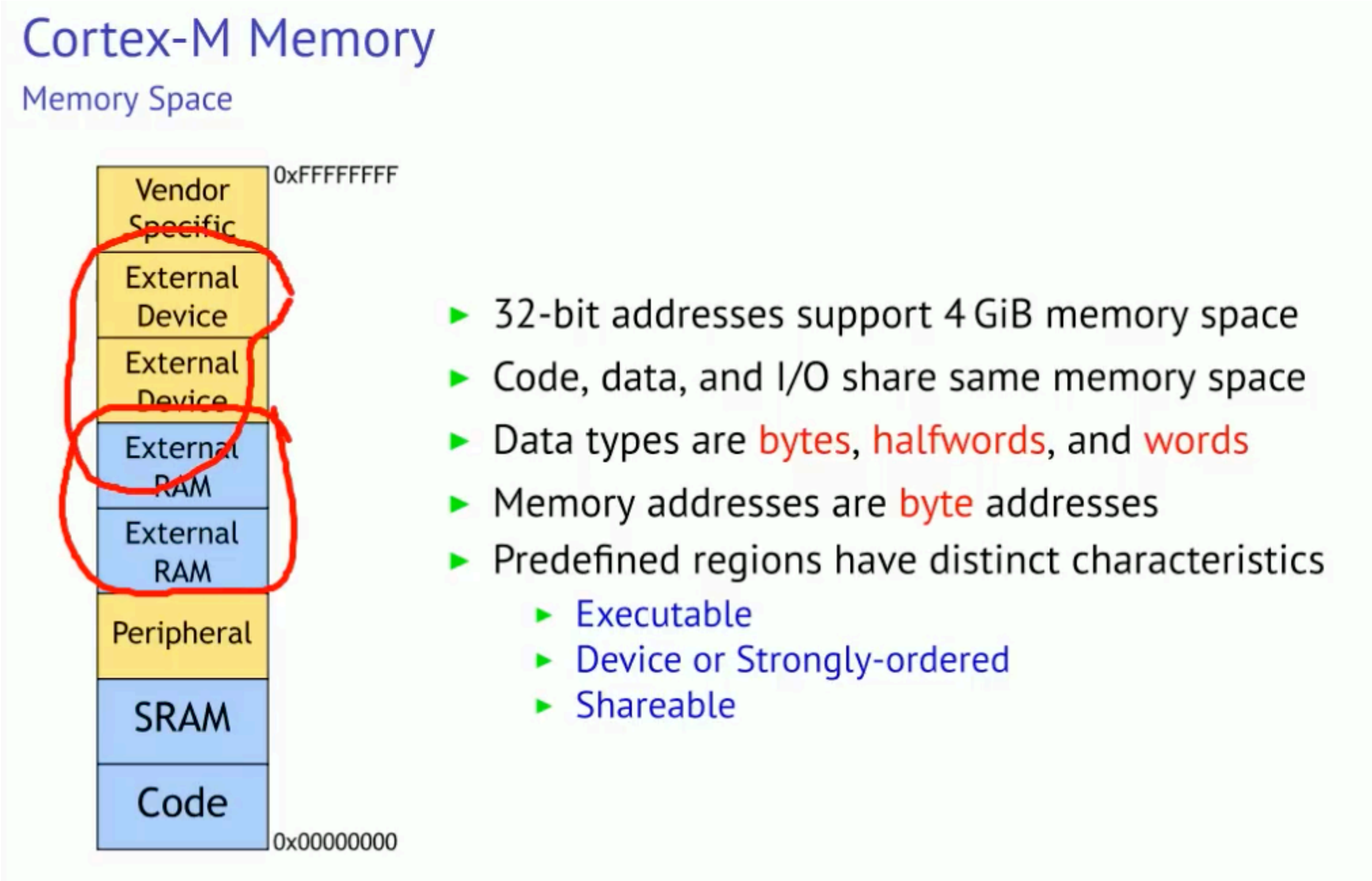
Sixteen generic 32-bit registers

- Thirteen are for general purposes
  - Can hold data or address
  - Data may be byte, halfword, or word
- Three have a special purpose
  - R13 is the stack pointer

+ one special purpose register, the (application) program status register (A)PSR:

N negative
Z zero
C carry
V overflow
Q saturation

An Example ISA: ARM Cortex 7



Cortex-M Memory

Memory Space

- 32-bit addresses support 4 GiB memory space
- Code, data, and I/O share same memory space
- Data types are bytes, halfwords, and words
- Memory addresses are byte addresses
- Predefined regions have distinct characteristics
  - Executable
  - Device or Strongly-ordered
  - Shareable

An Example ISA: ARM Cortex 7

An Example ISA: ARM Cortex 7

- Register Transfer Operations

An Example ISA: ARM Cortex 7

- Register Transfer Operations

An Example ISA: ARM Cortex 7

- Register Transfer Operations

0x00000011	R0
0x00000A00	R1
0xFFFFFFFB	R2
0xFEEDC0DE	R3
0x00000A00	R4
	R5
	R6
	R7
	R8
	R9
	R10
	R11
	R12
	R13
	R14

```
MOV   R0, #0x11
MOV   R1, #2560
MVN   R2, #4
MOVW  R3, #0xC0DE
MOVT  R3, #0xFEED
MOV   R4, R1
```

An Example ISA: ARM Cortex 7

An Example ISA: ARM Cortex 7

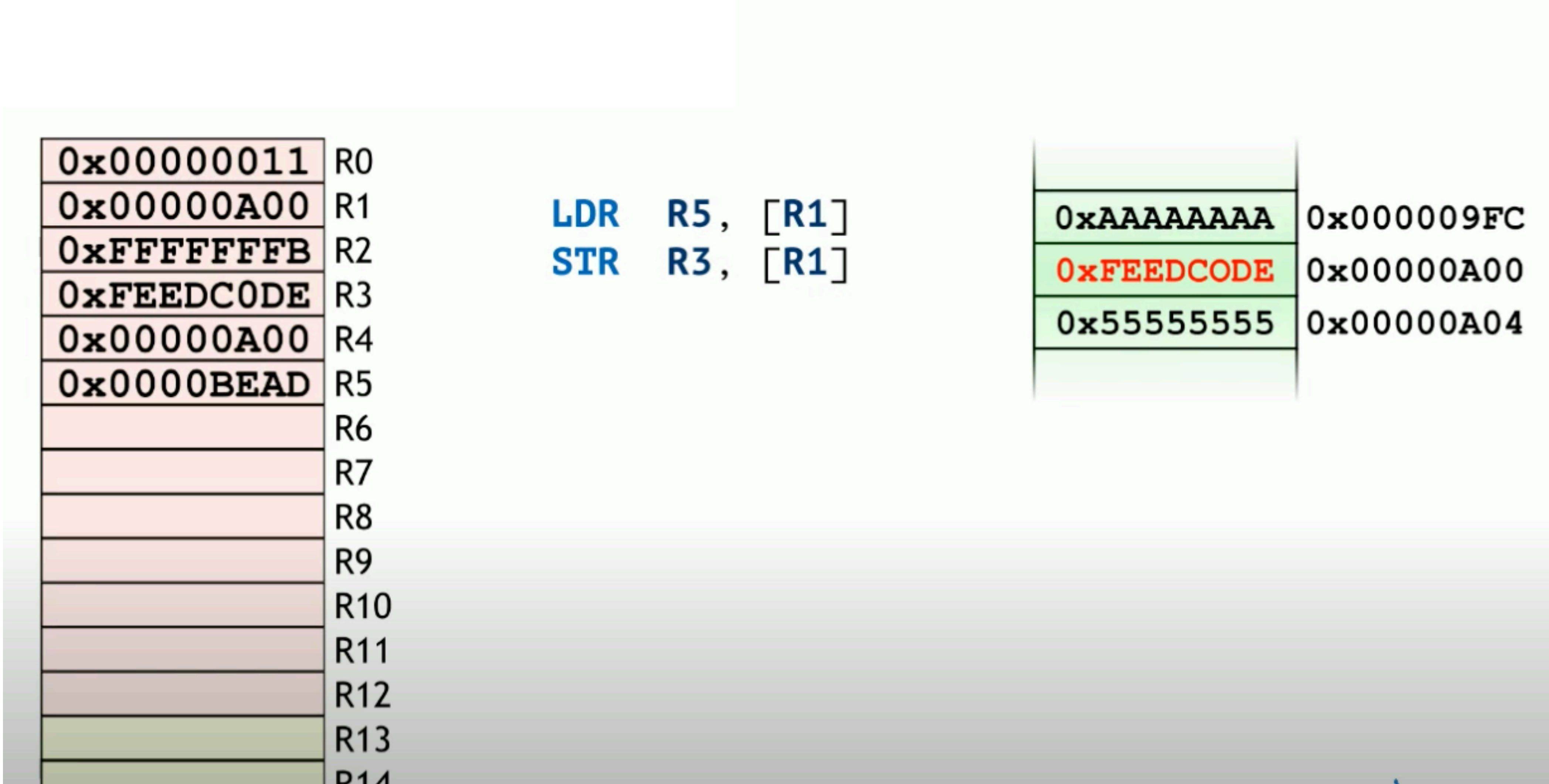
- Basic Load/Store operations

An Example ISA: ARM Cortex 7

- Basic Load/Store operations

An Example ISA: ARM Cortex 7

- Basic Load/Store operations



0x00000011	R0
0x00000A00	R1
0xFFFFFFFB	R2
0xFEEDC0DE	R3
0x00000A00	R4
0x0000BEAD	R5
	R6
	R7
	R8
	R9
	R10
	R11
	R12
	R13
	R14

```
LDR   R5, [R1]
STR   R3, [R1]
```

0xAAAAAAAA	0x000009FC
0xFEEDC0DE	0x00000A00
0x55555555	0x00000A04

An Example ISA: ARM Cortex 7

An Example ISA: ARM Cortex 7

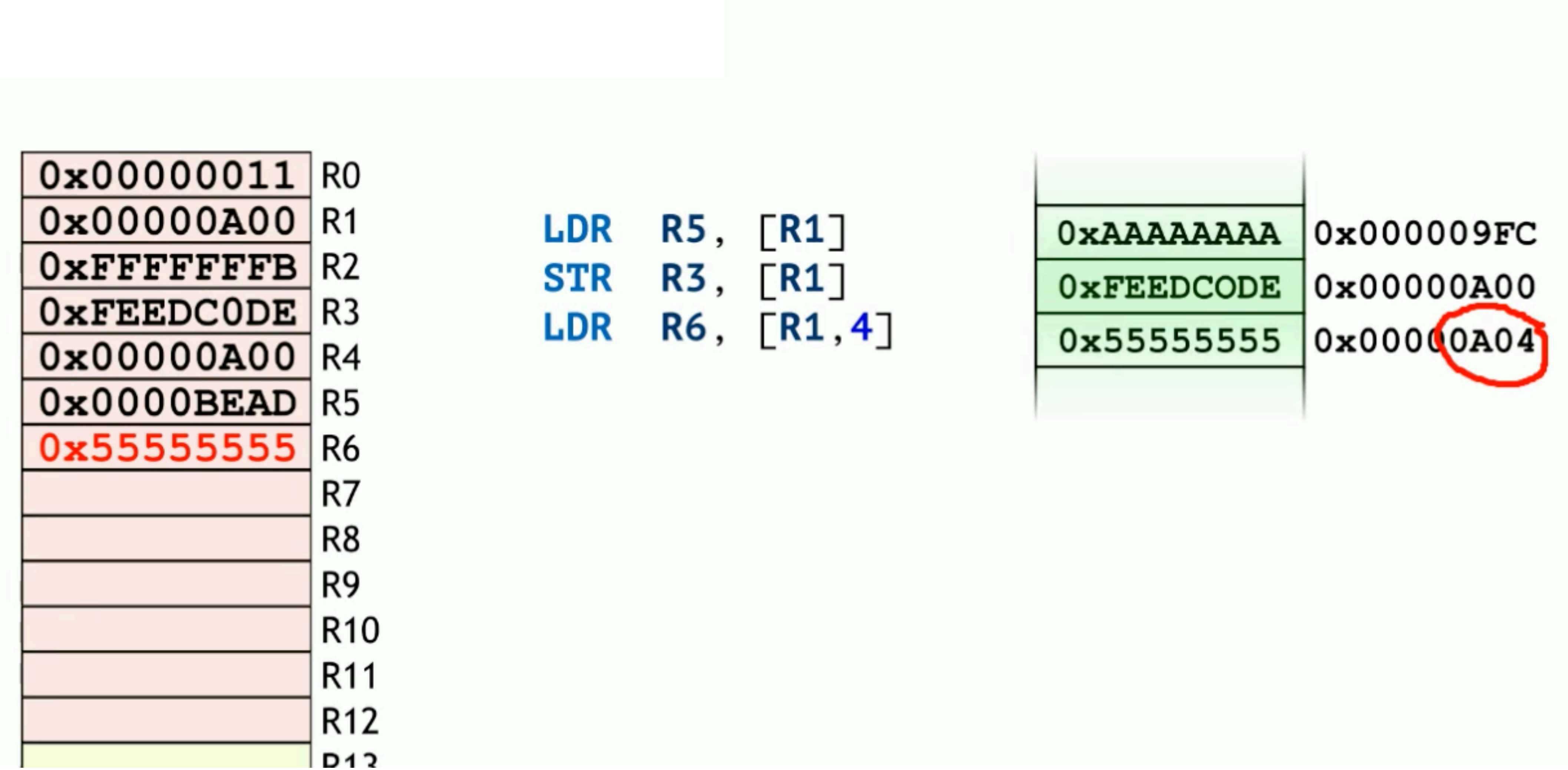
- Basic Load/Store with offsets (for arrays)

An Example ISA: ARM Cortex 7

- Basic Load/Store with offsets (for arrays)

An Example ISA: ARM Cortex 7

- Basic Load/Store with offsets (for arrays)



0x00000011	R0
0x00000A00	R1
0xFFFFFFFB	R2
0xFEEDC0DE	R3
0x00000A00	R4
0x0000BEAD	R5
0x55555555	R6
	R7
	R8
	R9
	R10
	R11
	R12
	R13

```
LDR   R5, [R1]
STR   R3, [R1]
LDR   R6, [R1,4]
```

0xAAAAAAAA	0x000009FC
0xFEEDC0DE	0x00000A00
0x55555555	0x00000A04

An Example ISA: ARM Cortex 7

An Example ISA: ARM Cortex 7

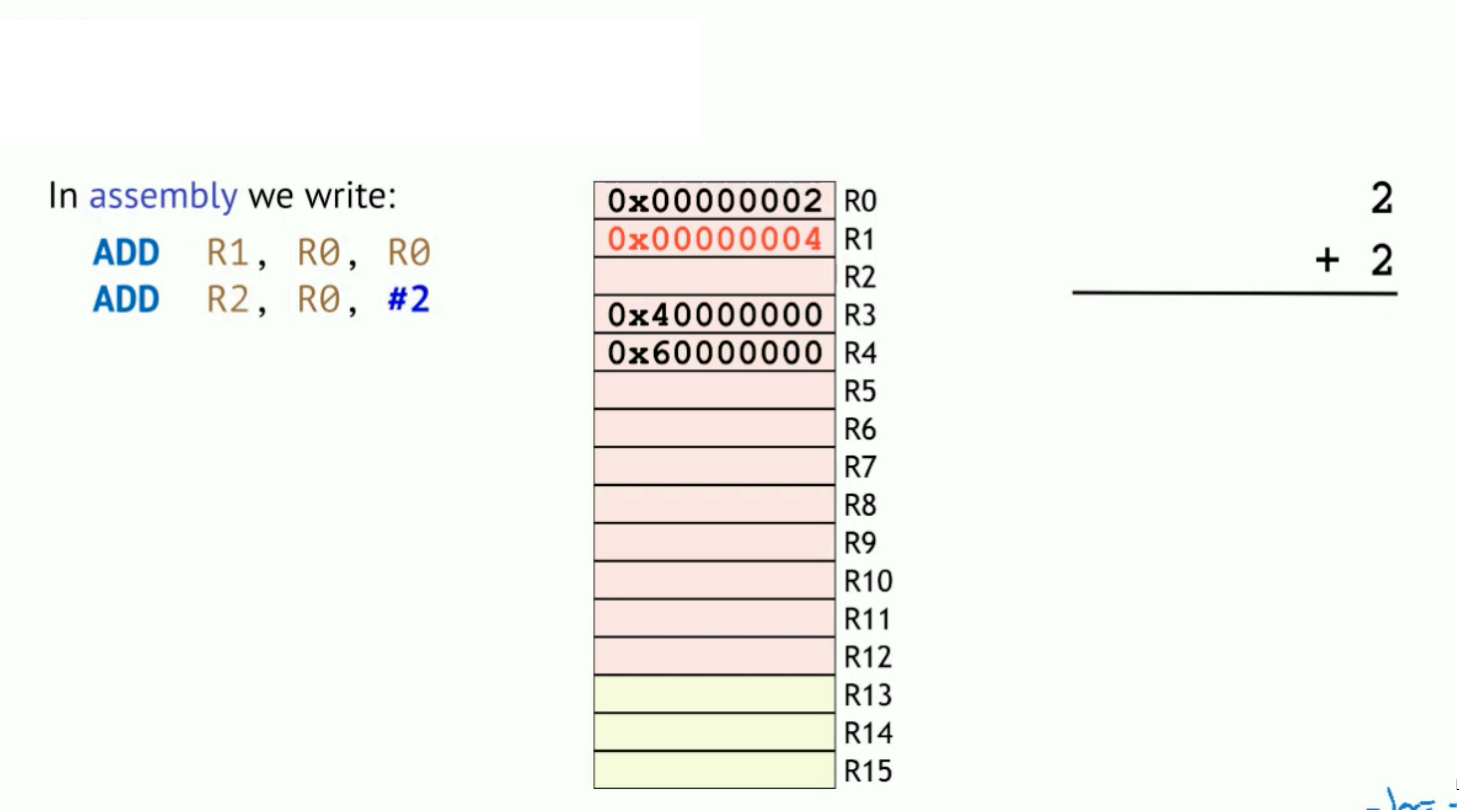
- Arithmetic Operators : addition

An Example ISA: ARM Cortex 7

- Arithmetic Operators : addition

An Example ISA: ARM Cortex 7

- Arithmetic Operators : addition



In assembly we write:

```
    ADD   R1, R0, R0
    ADD   R2, R0, #2
```

0x00000002	R0
0x00000004	R1
	R2
0x40000000	R3
0x60000000	R4
	R5
	R6
	R7
	R8
	R9
	R10
	R11
	R12
	R13
	R14
	R15

$$\begin{array}{r} 2 \\ +\ 2 \\ \hline \end{array}$$

An Example ISA: ARM Cortex 7

An Example ISA: ARM Cortex 7

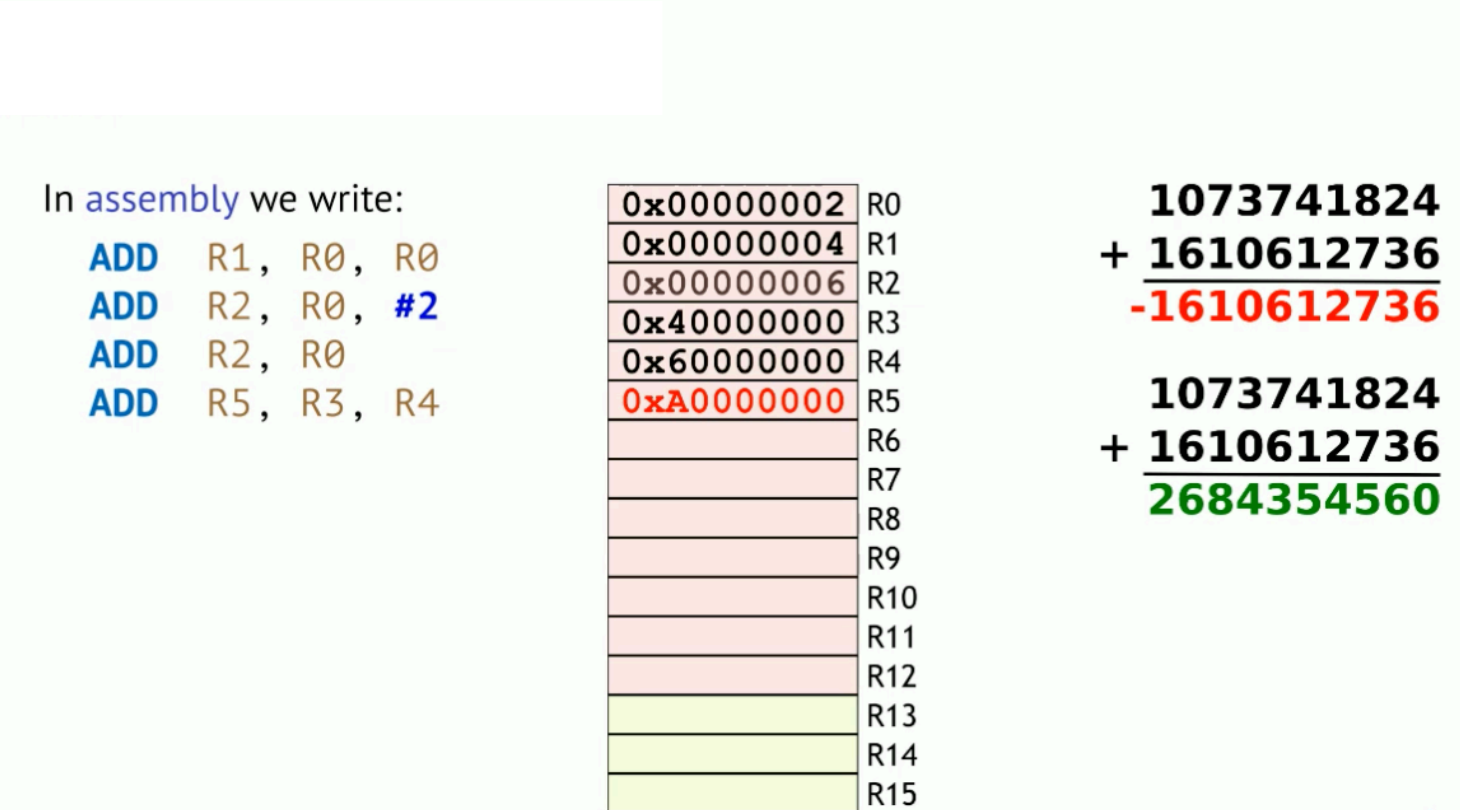
- Attention: 2s complement calculations !!!

An Example ISA: ARM Cortex 7

- Attention: 2s complement calculations !!!

An Example ISA: ARM Cortex 7

- Attention: 2s complement calculations !!!



In assembly we write:

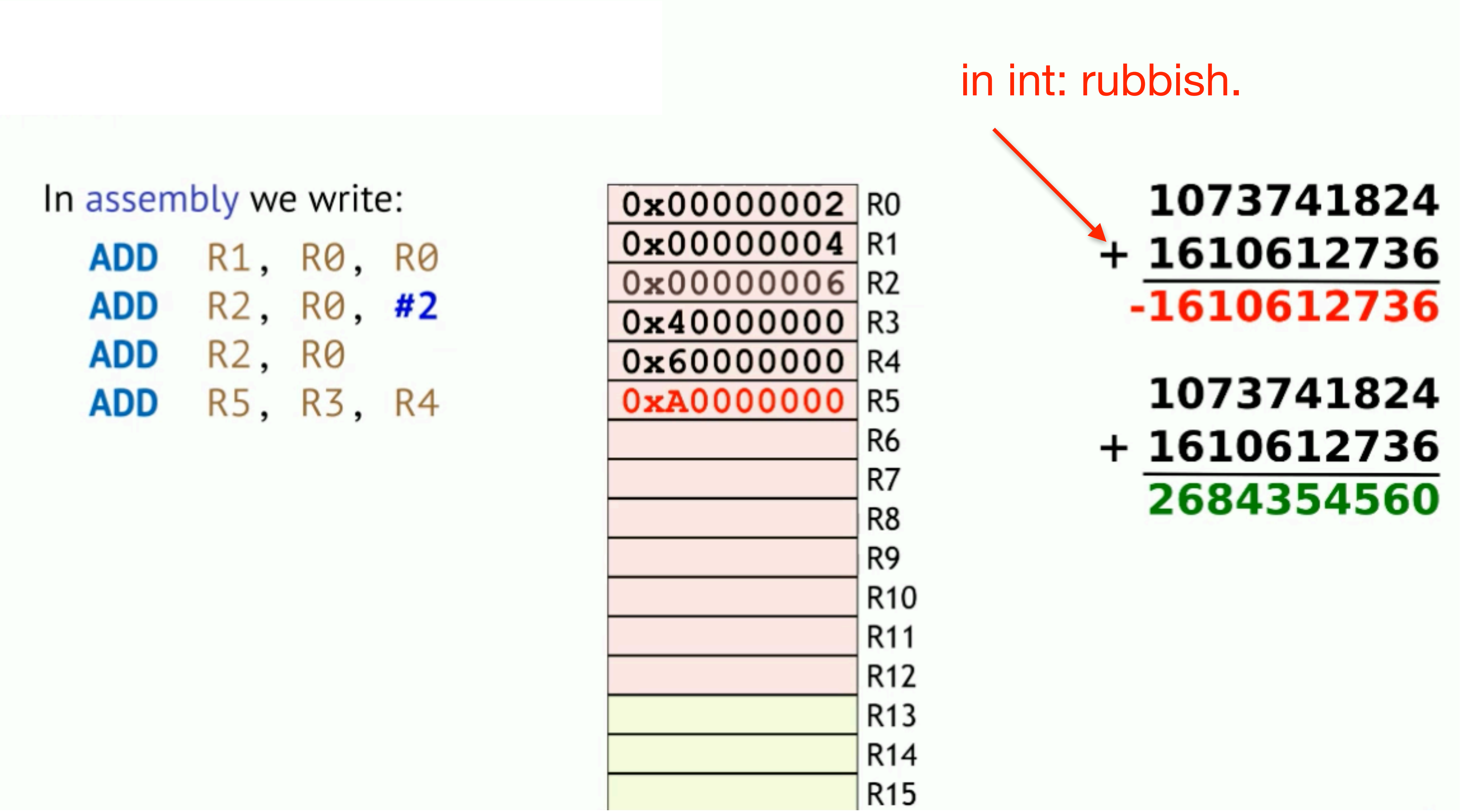
```
ADD   R1, R0, R0
ADD   R2, R0, #2
ADD   R2, R0
ADD   R5, R3, R4
```

0x00000002	R0
0x00000004	R1
0x00000006	R2
0x40000000	R3
0x60000000	R4
0xA0000000	R5
	R6
	R7
	R8
	R9
	R10
	R11
	R12
	R13
	R14
	R15

```
  1073741824
+ 1610612736
 -1610612736

  1073741824
+ 1610612736
  2684354560
```

An Example ISA: ARM Cortex 7

- Attention: 2s complement calculations !!!



in int: rubbish.

In assembly we write:

```
ADD   R1, R0, R0
ADD   R2, R0, #2
ADD   R2, R0
ADD   R5, R3, R4
```

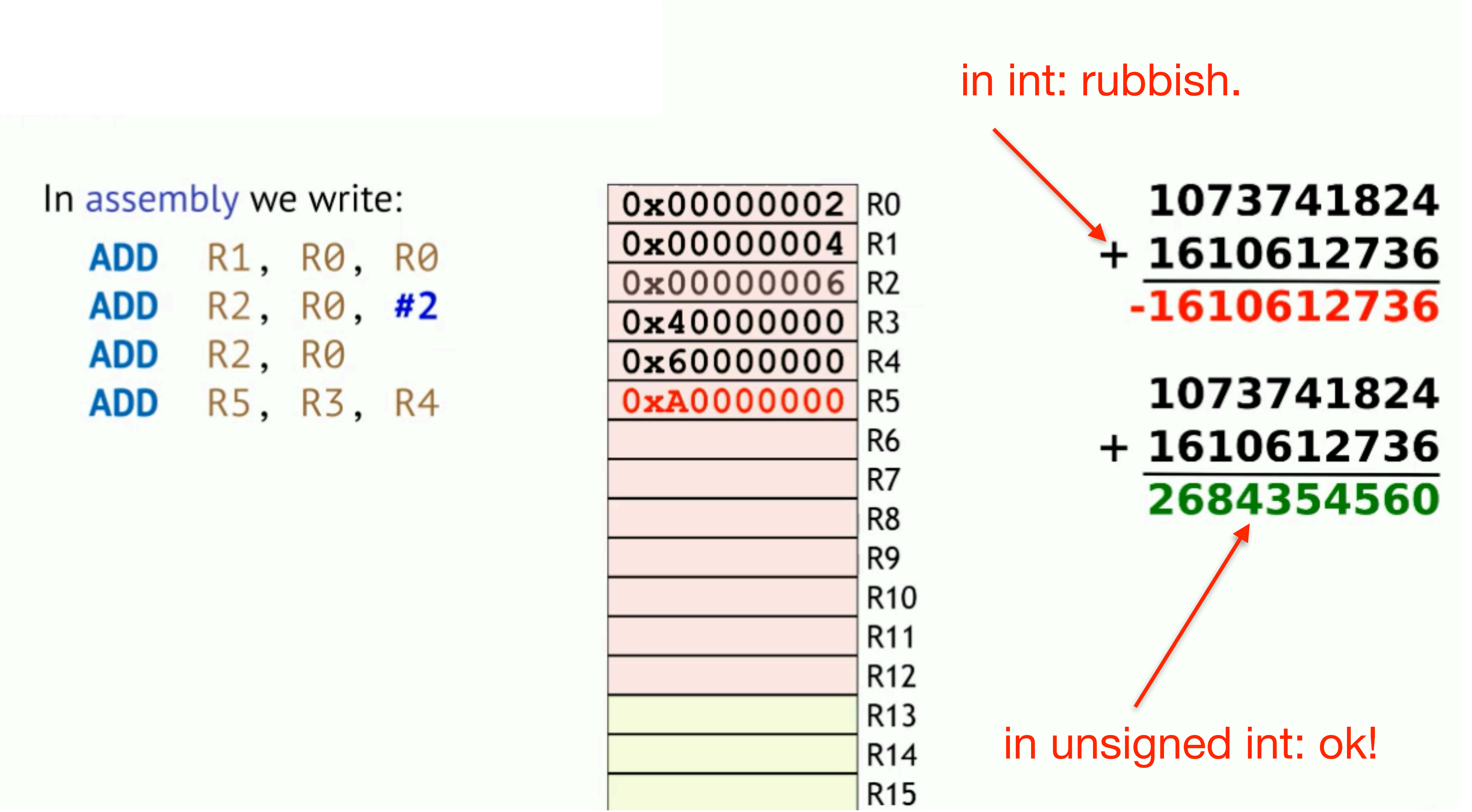
0x00000002	R0
0x00000004	R1
0x00000006	R2
0x40000000	R3
0x60000000	R4
0xA0000000	R5
	R6
	R7
	R8
	R9
	R10
	R11
	R12
	R13
	R14
	R15

$$\begin{array}{r} 1073741824 \\ +\ 1610612736 \\ \hline -1610612736 \end{array}$$

$$\begin{array}{r} 1073741824 \\ +\ 1610612736 \\ \hline 2684354560 \end{array}$$

An Example ISA: ARM Cortex 7

- Attention: 2s complement calculations !!!



In assembly we write:

```
ADD   R1, R0, R0
ADD   R2, R0, #2
ADD   R2, R0
ADD   R5, R3, R4
```

0x00000002	R0
0x00000004	R1
0x00000006	R2
0x40000000	R3
0x60000000	R4
0xA0000000	R5
	R6
	R7
	R8
	R9
	R10
	R11
	R12
	R13
	R14
	R15

in int: rubbish.

$$\begin{array}{r} 1073741824 \\ +\ 1610612736 \\ \hline -1610612736 \end{array}$$

$$\begin{array}{r} 1073741824 \\ +\ 1610612736 \\ \hline 2684354560 \end{array}$$

in unsigned int: ok!

An Example ISA: ARM Cortex 7

An Example ISA: ARM Cortex 7

- Arithmetic Operators : multiplication

An Example ISA: ARM Cortex 7

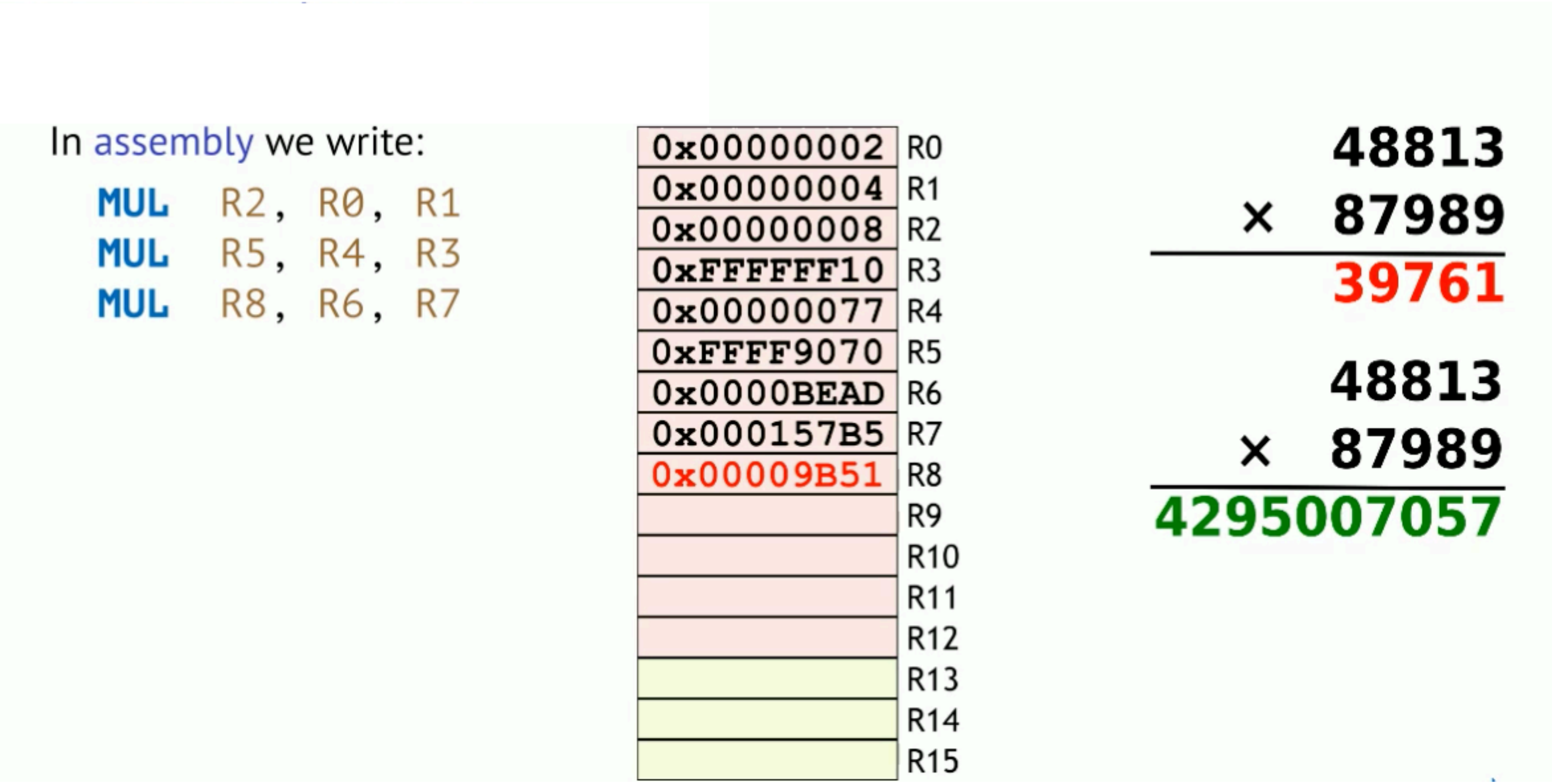
- Arithmetic Operators : multiplication

An Example ISA: ARM Cortex 7

- Arithmetic Operators : multiplication

An Example ISA: ARM Cortex 7

- Arithmetic Operators : multiplication



In assembly we write:

```
MUL   R2, R0, R1
MUL   R5, R4, R3
MUL   R8, R6, R7
```

0x00000002	R0
0x00000004	R1
0x00000008	R2
0xFFFFFF10	R3
0x00000077	R4
0xFFFF9070	R5
0x0000BEAD	R6
0x000157B5	R7
0x00009B51	R8
	R9
	R10
	R11
	R12
	R13
	R14
	R15

$$\begin{array}{r} 48813 \\ \times\ \ 87989 \\ \hline 39761 \end{array}$$

$$\begin{array}{r} 48813 \\ \times\ \ 87989 \\ \hline 4295007057 \end{array}$$

An Example ISA: ARM Cortex 7

An Example ISA: ARM Cortex 7

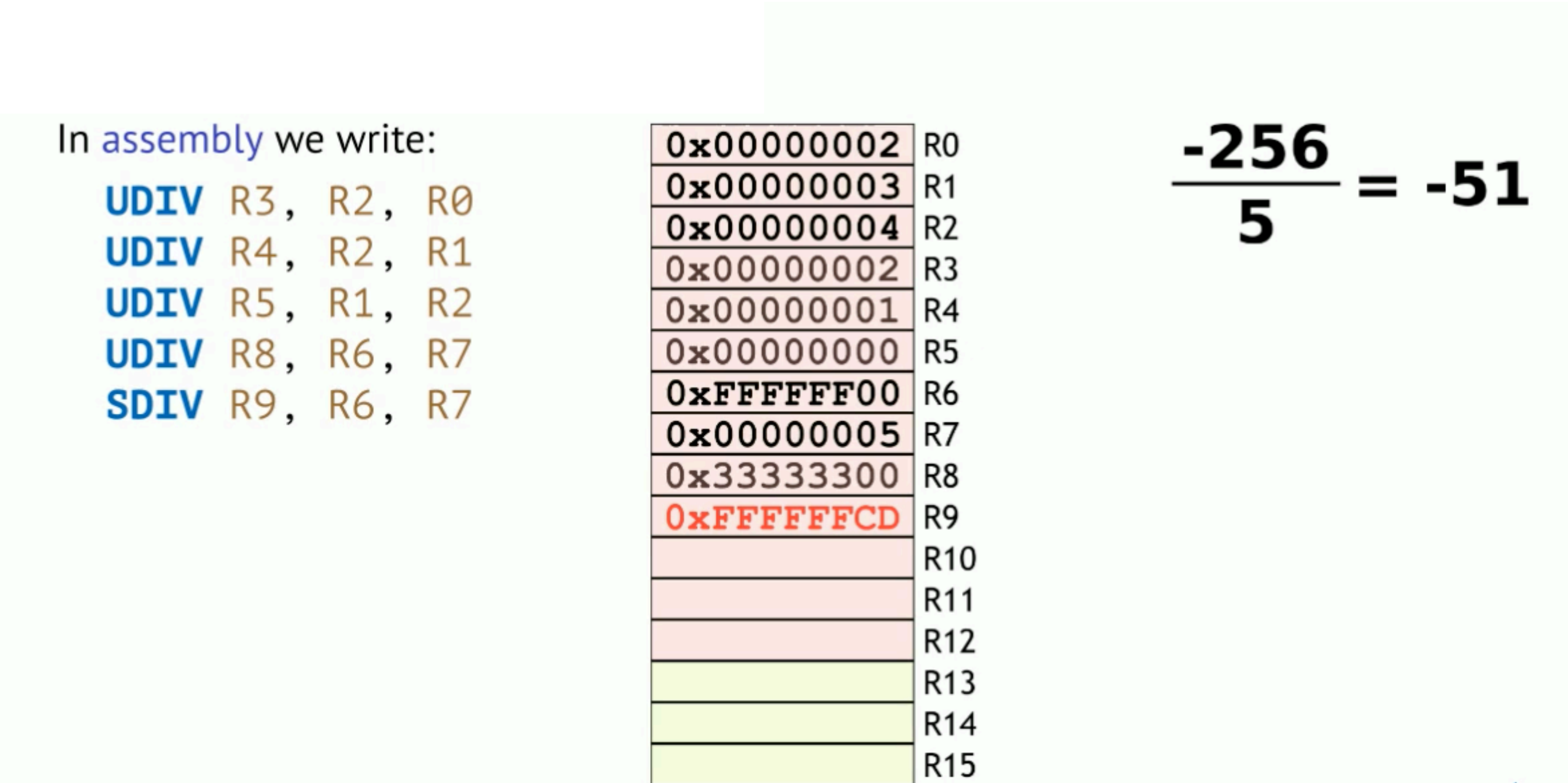
- Arithmetic Operators :
  2 divisions: signed SDIV et unsigned UDIV.

An Example ISA: ARM Cortex 7

- Arithmetic Operators :
  2 divisions: signed SDIV et unsigned UDIV.

An Example ISA: ARM Cortex 7

- Arithmetic Operators :
  2 divisions: signed SDIV et unsigned UDIV.



In assembly we write:

```
UDIV R3, R2, R0
UDIV R4, R2, R1
UDIV R5, R1, R2
UDIV R8, R6, R7
SDIV R9, R6, R7
```

0x00000002	R0
0x00000003	R1
0x00000004	R2
0x00000002	R3
0x00000001	R4
0x00000000	R5
0xFFFFFF00	R6
0x00000005	R7
0x33333300	R8
0xFFFFFFCD	R9
	R10
	R11
	R12
	R13
	R14
	R15

$$\frac{-256}{5} = -51$$

An Example ISA: ARM Cortex 7

An Example ISA: ARM Cortex 7

- Control Flow Operation
  (Ops that influence R15 (pc) in a way)

An Example ISA: ARM Cortex 7

- Control Flow Operation
  (Ops that influence R15 (pc) in a way)

- Branch (B <label>),
  Branch indirect (BX <Rn>),

An Example ISA: ARM Cortex 7

- Control Flow Operation
  (Ops that influence R15 (pc) in a way)

- Branch (B <label>),
  Branch indirect (BX <Rn>),

- Conditional branches
  BLE <label>, BNE <label>
  (depends on status register APSR)

An Example ISA: ARM Cortex 7

- Control Flow Operation
  (Ops that influence R15 (pc) in a way)

- Branch (B <label>),
  Branch indirect (BX <Rn>),

- Conditional branches

  BLE <label>, BNE <label>

  (depends on status register APSR)

- Compare and Branch: CBZ <Rn>,<label>:

```
CMP    Rn, #0
BEQ    <label>
```

An Example ISA: ARM Cortex 7

- Control Flow Operation
  (Ops that influence R15 (pc) in a way)

- Branch (B <label>),
  Branch indirect (BX <Rn>),

- Conditional branches

  BLE <label>, BNE <label>

  (depends on status register APSR)

- Compare and Branch: CBZ <Rn>,<label>:

```
CMP    Rn, #0
BEQ    <label>
```

An Example ISA: ARM Cortex 7

- Control Flow Operation
  (Ops that influence R15 (pc) in a way)
- Branch (B <label>),
  Branch indirect (BX <Rn>),
- Conditional branches

  BLE <label>, BNE <label>

  (depends on status register APSR)
- Compare and Branch: CBZ <Rn>,<label>:

```
CMP    Rn, #0
BEQ    <label>
```

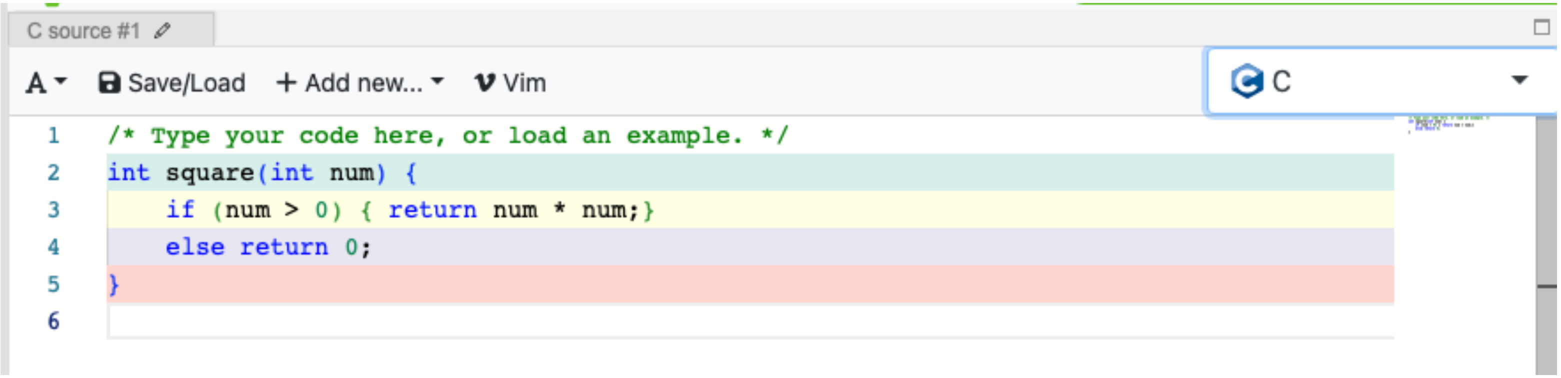
Does not change APSR !

Putting Together an Example:

(generated by 'Compiler Explorer': https://godbolt.org/)

Putting Together an Example:

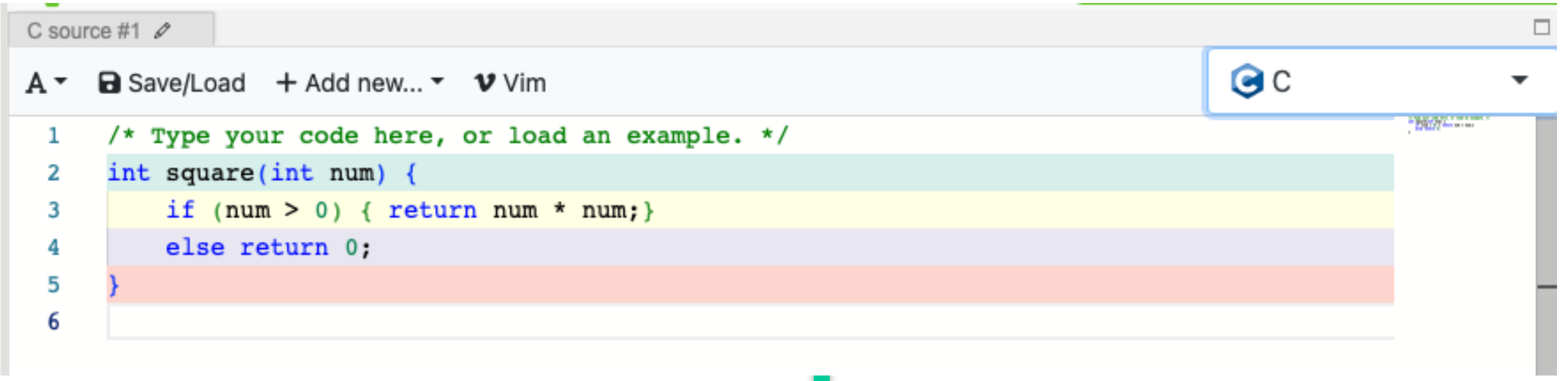
(generated by 'Compiler Explorer': https://godbolt.org/)



```c
/* Type your code here, or load an example. */
int square(int num) {
    if (num > 0) { return num * num;}
    else return 0;
}
```

Putting Together an Example:

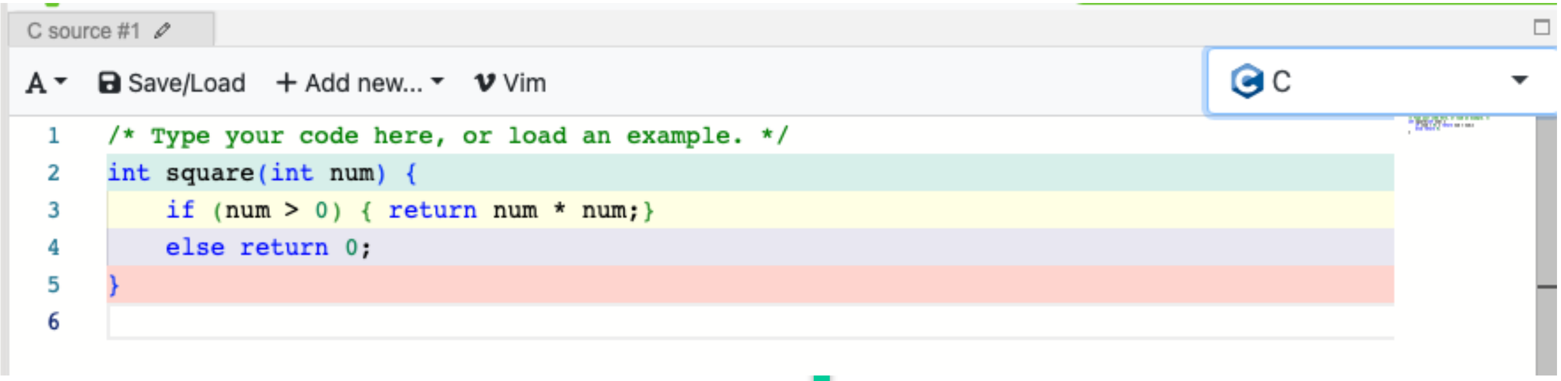
(generated by `Compiler Explorer': https://godbolt.org/)



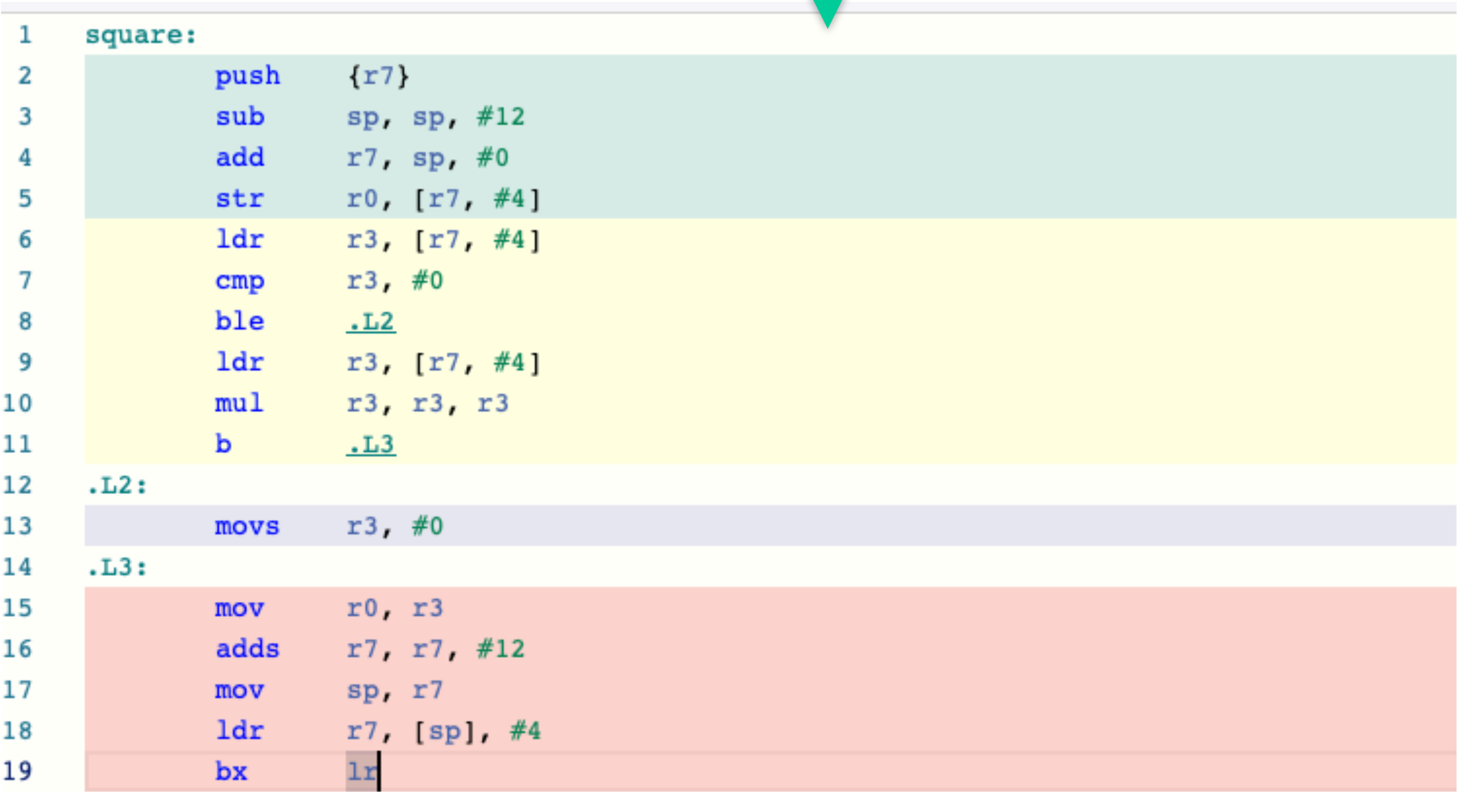
```c
/* Type your code here, or load an example. */
int square(int num) {
    if (num > 0) { return num * num;}
    else return 0;
}

```

Putting Together an Example:

(generated by 'Compiler Explorer': https://godbolt.org/)



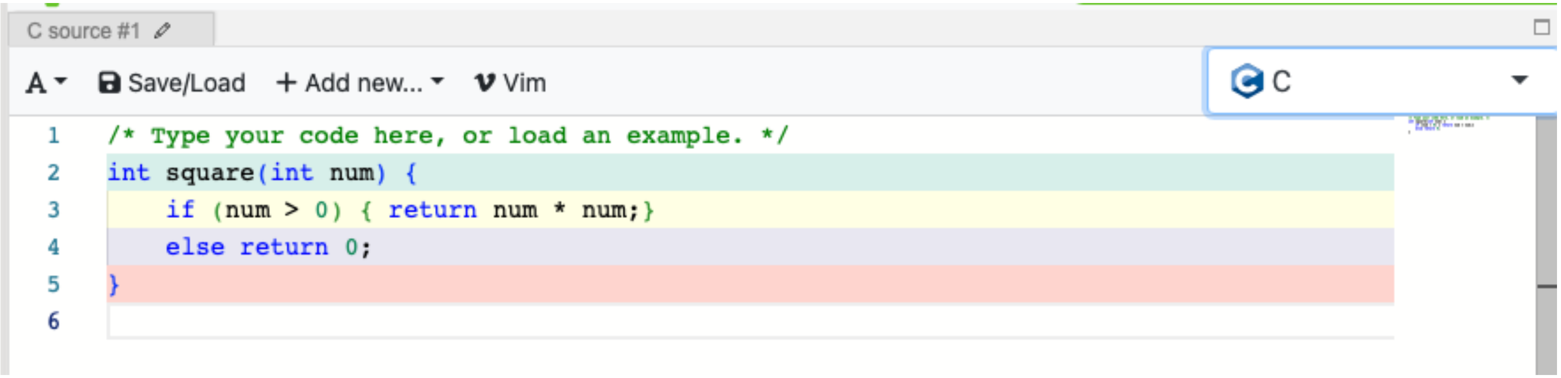
```c
/* Type your code here, or load an example. */
int square(int num) {
    if (num > 0) { return num * num;}
    else return 0;
}
```



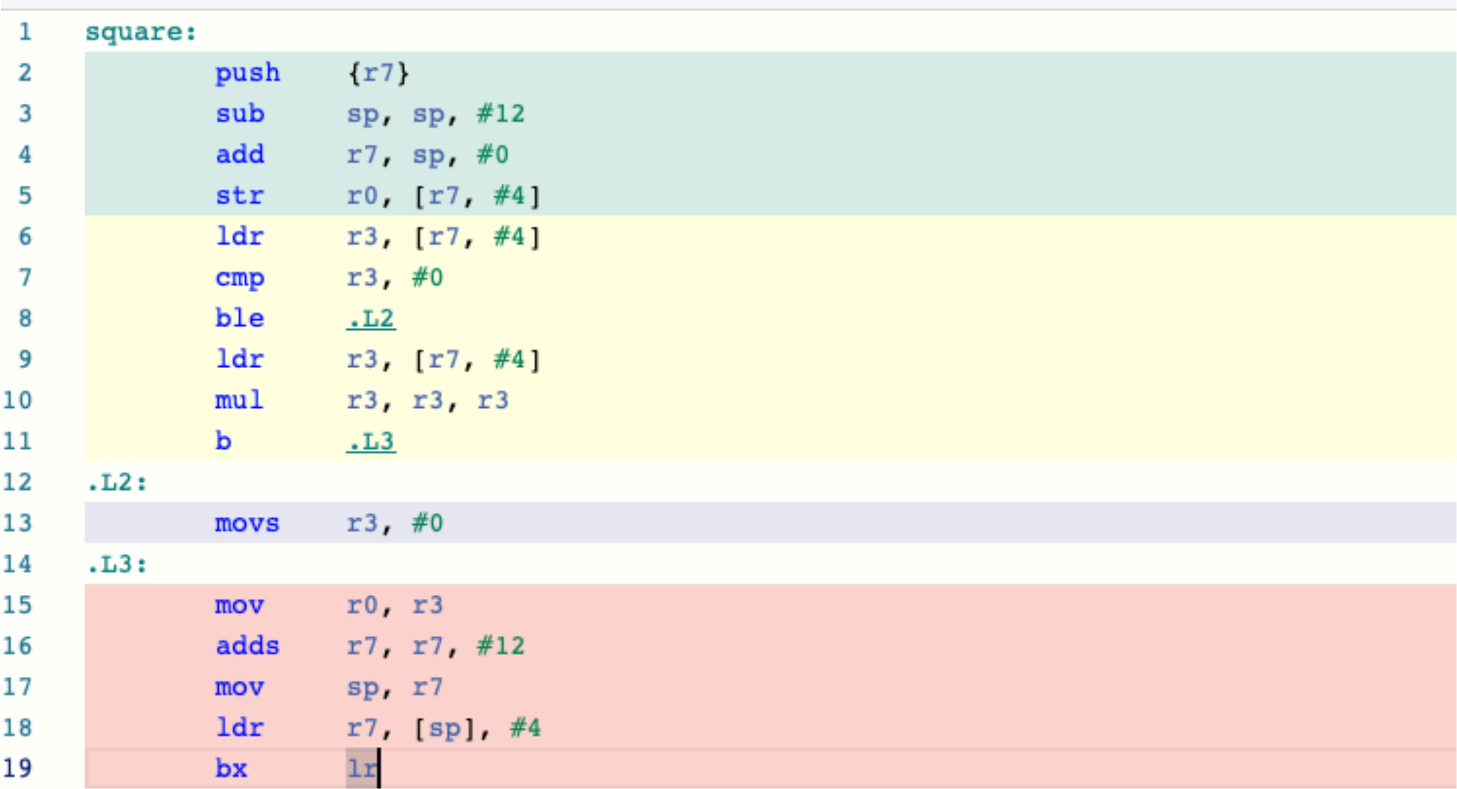
```
square:
        push    {r7}
        sub     sp, sp, #12
        add     r7, sp, #0
        str     r0, [r7, #4]
        ldr     r3, [r7, #4]
        cmp     r3, #0
        ble     .L2
        ldr     r3, [r7, #4]
        mul     r3, r3, r3
        b       .L3
.L2:
        movs    r3, #0
.L3:
        mov     r0, r3
        adds    r7, r7, #12
        mov     sp, r7
        ldr     r7, [sp], #4
        bx      lr
```

Putting Together an Example:

(generated by 'Compiler Explorer': https://godbolt.org/)



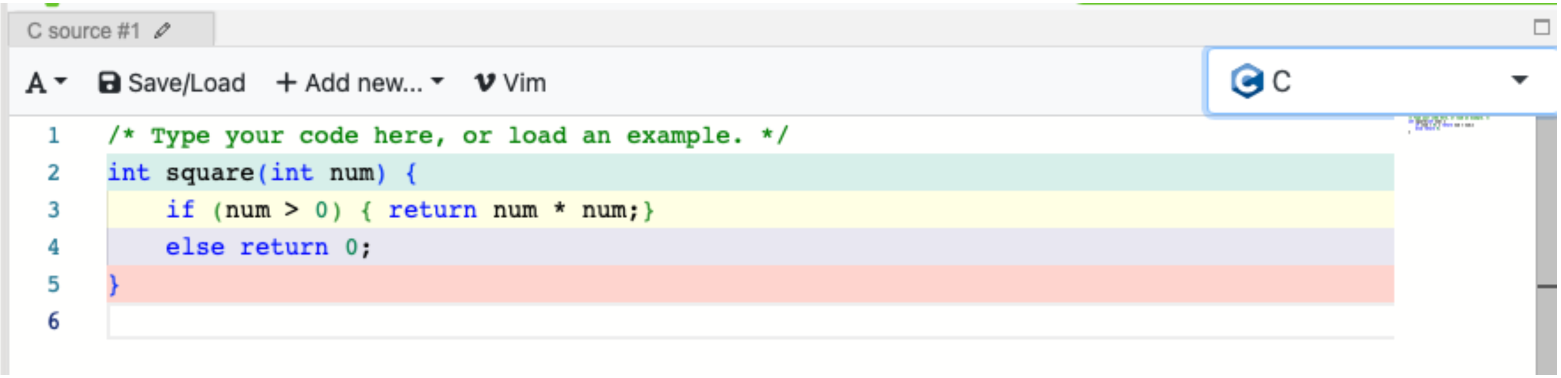
```c
/* Type your code here, or load an example. */
int square(int num) {
    if (num > 0) { return num * num;}
    else return 0;
}
```



```
square:
        push    {r7}
        sub     sp, sp, #12
        add     r7, sp, #0
        str     r0, [r7, #4]
        ldr     r3, [r7, #4]
        cmp     r3, #0
        ble     .L2
        ldr     r3, [r7, #4]
        mul     r3, r3, r3
        b       .L3
.L2:
        movs    r3, #0
.L3:
        mov     r0, r3
        adds    r7, r7, #12
        mov     sp, r7
        ldr     r7, [sp], #4
        bx      lr
```

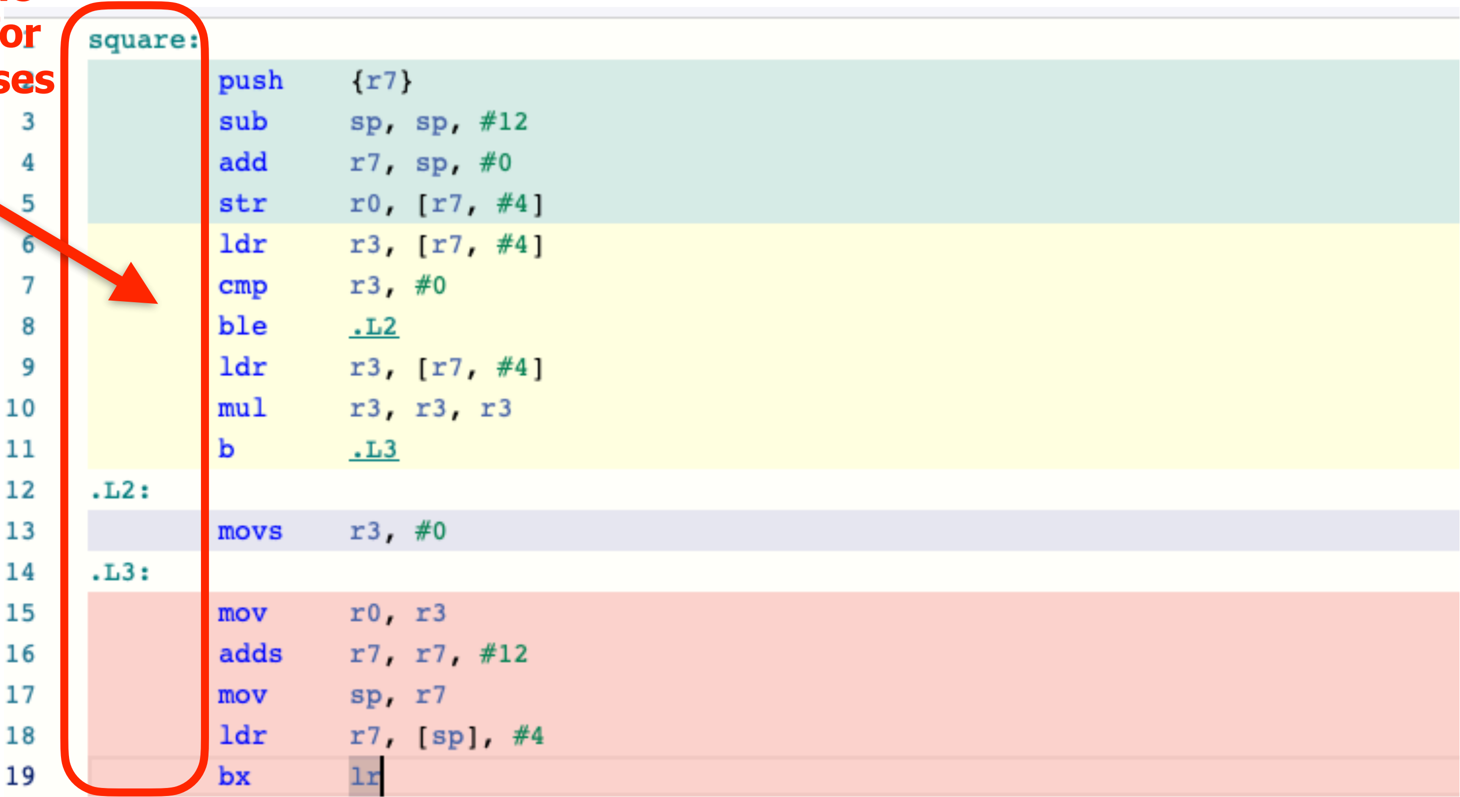
Putting Together an Example:

(generated by 'Compiler Explorer': https://godbolt.org/)



```
/* Type your code here, or load an example. */
int square(int num) {
    if (num > 0) { return num * num;}
    else return 0;
}
```

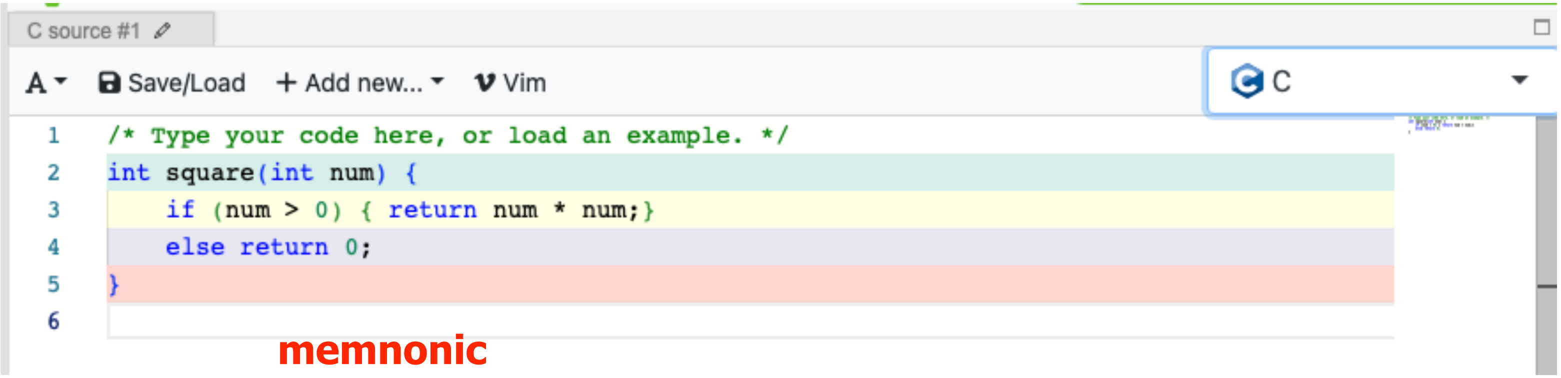
symbolic labels for addresses



```
square:
        push    {r7}
        sub     sp, sp, #12
        add     r7, sp, #0
        str     r0, [r7, #4]
        ldr     r3, [r7, #4]
        cmp     r3, #0
        ble     .L2
        ldr     r3, [r7, #4]
        mul     r3, r3, r3
        b       .L3
.L2:
        movs    r3, #0
.L3:
        mov     r0, r3
        adds    r7, r7, #12
        mov     sp, r7
        ldr     r7, [sp], #4
        bx      lr
```

Putting Together an Example:

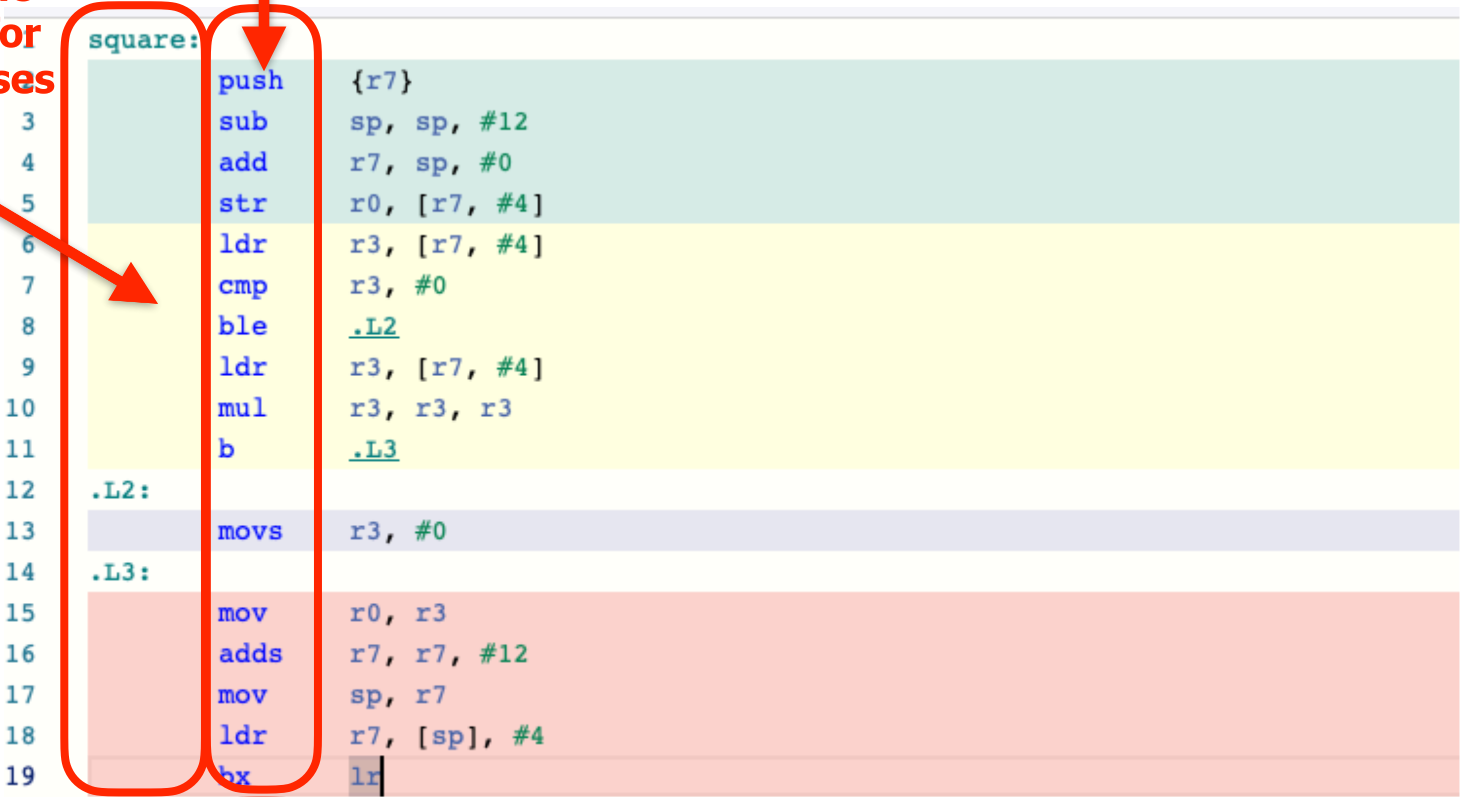
(generated by 'Compiler Explorer': https://godbolt.org/)



```c
/* Type your code here, or load an example. */
int square(int num) {
    if (num > 0) { return num * num;}
    else return 0;
}
```

memnonic operation codes

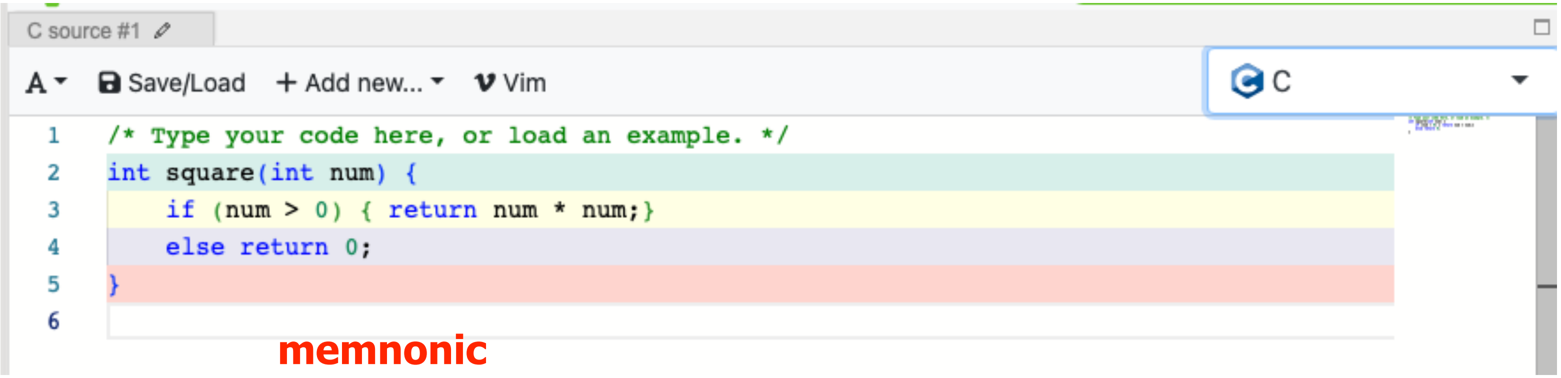
symbolic labels for addresses



```
square:
        push    {r7}
        sub     sp, sp, #12
        add     r7, sp, #0
        str     r0, [r7, #4]
        ldr     r3, [r7, #4]
        cmp     r3, #0
        ble     .L2
        ldr     r3, [r7, #4]
        mul     r3, r3, r3
        b       .L3
.L2:
        movs    r3, #0
.L3:
        mov     r0, r3
        adds    r7, r7, #12
        mov     sp, r7
        ldr     r7, [sp], #4
        bx      lr
```

Putting Together an Example:

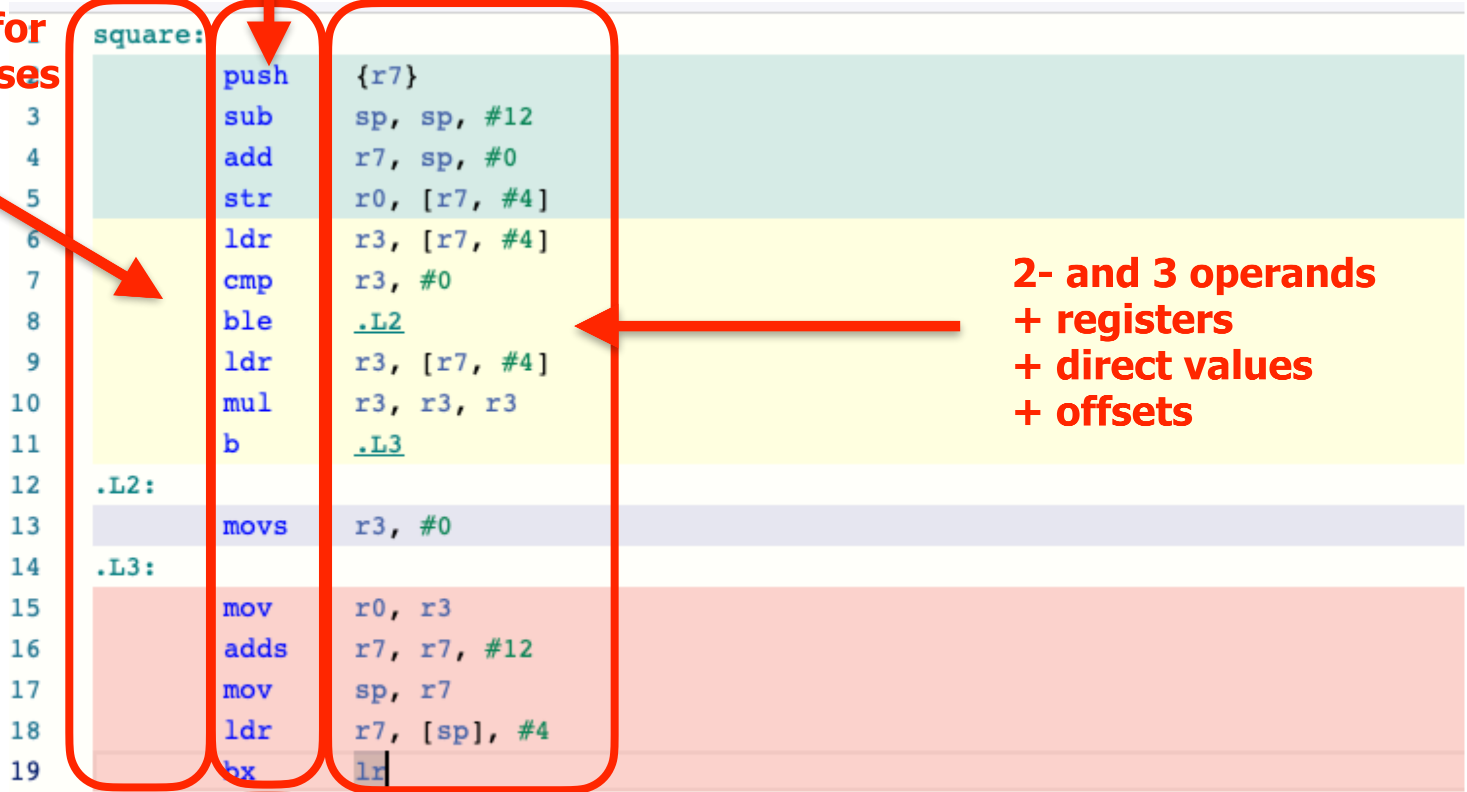
(generated by 'Compiler Explorer': https://godbolt.org/)



```c
/* Type your code here, or load an example. */
int square(int num) {
    if (num > 0) { return num * num;}
    else return 0;
}
```

memnonic operation codes

symbolic labels for addresses



```
square:
        push    {r7}
        sub     sp, sp, #12
        add     r7, sp, #0
        str     r0, [r7, #4]
        ldr     r3, [r7, #4]
        cmp     r3, #0
        ble     .L2
        ldr     r3, [r7, #4]
        mul     r3, r3, r3
        b       .L3
.L2:
        movs    r3, #0
.L3:
        mov     r0, r3
        adds    r7, r7, #12
        mov     sp, r7
        ldr     r7, [sp], #4
        bx      lr
```

2- and 3 operands
+ registers
+ direct values
+ offsets

Specifying Code Generation

Specifying Code Generation

- Specifying the boolean expression source lang:

```
“d = ((a) && ((b) || c))”;
```

Specifying Code Generation

- Specifying the boolean expression source lang:

  “d = ((a) && ((b) || c))”;

- AST of boolean expression language:

```
datatype expr = LVAR string
              | And   expr expr
              | Or expr expr
              | Not expr
```

Specifying Code Generation

- Specifying the boolean expression source lang:

“d = ((a) && ((b) || c))”;

- AST of boolean expression language:

```
datatype expr = LVAR string
              | And   expr expr
              | Or expr expr
              | Not expr
```

- Assumptions:

Specifying Code Generation

- Specifying the boolean expression source lang:

  “d = ((a) && ((b) || c))”;

- AST of boolean expression language:

```
datatype expr = LVAR string
              | And   expr expr
              | Or expr expr
              | Not expr
```

- Assumptions:
  - local vars already stored in registers (marked in a precomputed environment)

Specifying Code Generation

- Specifying the boolean expression source lang:

  “d = ((a) && ((b) || c))”;

- AST of boolean expression language:

```
datatype expr = LVAR string
              | And   expr expr
              | Or expr expr
              | Not expr
```

- Assumptions:
  - local vars already stored in registers (marked in a precomputed environment)
  - all LVARs fit into the registers

Specifying Code Generation

- Specifying the boolean expression source lang:

  "d = ((a) && ((b) || c))";

- AST of boolean expression language:

```
datatype expr = LVAR string
              | And   expr expr
              | Or expr expr
              | Not expr
```

- Assumptions:
  - local vars already stored in registers (marked in a precomputed environment)
  - all LVARs fit into the registers
  - result in register r7

Specifying Code Generation

- Specifying the boolean expression source lang:

  “d = ((a) && ((b) || c))”;

- AST of boolean expression language:

```
datatype expr = LVAR string
              | And   expr expr
              | Or expr expr
              | Not expr
```

- Assumptions:
  - local vars already stored in registers (marked in a precomputed environment)
  - all LVARs fit into the registers
  - result in register r7

Specifying Code Generation

Specifying Code Generation

- Specifying the target assembler AST

```
datatype reg = r0 | r1 | r2 | r3 | r4 | r5 | r6 | r7 | sp | lr | pc

datatype opcode = MOV | MVN | MOVW | MOVT | LDR | STR
                | ADD | SUB | MUL | UDIV | SDIV
                | B | BX | BLE | BEQ | BNQ | CBX | CMP

datatype varg = none                         (“---")
              | direct nat                   ("# _")
              | register2 reg reg            ("_,_")
              | register_indirect reg        ("[_]")
              | register_indirect_offset reg nat ("[_,_]")

datatype com = label nat ("⟨L _⟩”)
             | branch opcode nat ("⟨_ L _⟩")
             | instruction opcode reg varg ("⟨_,_,_⟩")
```

Specifying Code Generation

- Specifying the target assembler AST

```
datatype reg = r0 | r1 | r2 | r3 | r4 | r5 | r6 | r7 | sp | lr | pc

datatype opcode = MOV | MVN | MOVW | MOVT | LDR | STR
                | ADD | SUB | MUL | UDIV | SDIV
                | B | BX | BLE | BEQ | BNQ | CBX | CMP

datatype varg = none                                (“---")
              | direct nat                          ("# _")
              | register2 reg reg                   ("_,_")
              | register_indirect reg               ("[_]")
              | register_indirect_offset reg nat ("[_,_]")

datatype com = label nat ("⟨L _⟩”)
             | branch opcode nat ("⟨_ L _⟩")
             | instruction opcode reg varg ("⟨_,_,_⟩")
```

- SAMPLE: [⟨MOV,r3,#5⟩,⟨ADD,r5,r3,r4⟩,⟨L 2⟩,⟨ADD,r5,r3,r4⟩,⟨BLE,r5,[r3,4]⟩]

Specifying Code Generation

Specifying Code Generation

- Specifying the translation

Specifying Code Generation

- Specifying the translation
  - inherited and synthesised attributes

Env_{in} : expr ⇒ (string ⇒ reg)

Lab_{in} : expr ⇒ nat

Lab_{out} : expr ⇒ nat

Code_{out} : expr ⇒ com list

Specifying Code Generation

- Specifying the translation
  - inherited and synthesised attributes

    Env_{in} : expr ⇒ (string ⇒ reg)

    Lab_{in} : expr ⇒ nat

    Lab_{out} : expr ⇒ nat

    $Code_{out}$: expr ⇒ com list

  - Attribution for the case of a local variable:

    e = LVAR a:

    $Lab_{out}(e) = Lab_{in}(e)$
    $Code_{out}\ (e) = [\langle MOV, r7, Env_{in}\ (e)(a)\rangle]$

Specifying Code Generation

- Specifying the translation
  - inherited and synthesised attributes

    $Env_{in} : expr \Rightarrow (string \Rightarrow reg)$

    $Lab_{in} : expr \Rightarrow nat$

    $Lab_{out} : expr \Rightarrow nat$

    $Code_{out} : expr \Rightarrow com\ list$

  - Attribution for the case of a local variable:

    e = LVAR a:

    $Lab_{out}(e) = Lab_{in}(e)$

    $Code_{out}(e) = [\langle MOV, r7, Env_{in}(e)(a)\rangle]$

Specifying Code Generation

Specifying Code Generation

- Specifying the translation

Specifying Code Generation

- Specifying the translation
  - Attribution for the case of an And:

e = And a b :

$$
\begin{aligned}
&Lab_{out}(e) = Lab_{in}(b) \wedge \\
&Lab_{in}(b) = Lab_{in}(e)+1 \wedge \\
&Env_{in}(a) = Env_{in}(e) \wedge \\
&Env_{in}(b) = Env_{in}(e) \wedge \\
&Code_{out}(e) = Code_{out}(a) \\
&\qquad @ [\langle CMP,r7,\#0\rangle] \\
&\qquad @ [\langle BEQ\ L\ Lab_{in}(e)+1\rangle] \\
&\qquad @\ Code_{out}(b) \\
&\qquad @ [\langle L\ Lab_{in}(e) + 1\rangle]
\end{aligned}
$$

Specifying Code Generation

- Specifying the translation
  - Attribution for the case of an And:

e = And a b :

$Lab_{out}(e) = Lab_{in}(b) \wedge$

$Lab_{in}(b) = Lab_{in}(e)+1 \wedge$

$Env_{in}(a) = Env_{in}(e) \wedge$

$Env_{in}(b) = Env_{in}(e) \wedge$

$Code_{out}(e) = Code_{out}(a)$

$\quad @ [\langle CMP,r7,\#0\rangle]$

$\quad @ [\langle BEQ\ L\ Lab_{in}(e)+1\rangle]$

$\quad @ Code_{out}(b)$

$\quad @ [\langle L\ Lab_{in}(e) + 1\rangle]$

Specifying Code Generation

Specifying Code Generation

- Specifying the translation

Specifying Code Generation

- Specifying the translation
  - Attribution for the case of an Or:

e = Or a b:

$Lab_{out}(e) = Lab_{in}(b) \wedge$

Lab⇩i⇩n(b) $= Lab_{in}(e)+1 \wedge$

Env⇩i⇩n(a) $= Env_{in}(e) \wedge$

Env⇩i⇩n(b) $= Env_{in}(e) \wedge$

$Code_{out}(e) = Code_{out}\ (a)$

@ [⟨CMP,r7,#0⟩]

@ [⟨BNQ L $Lab_{in}(e)+1$⟩]

@ $Code_{out}\ (b)$

@ [⟨L $Lab_{in}(e) + 1$⟩]

Specifying Code Generation

- Specifying the translation
  - Attribution for the case of an Or:

e = Or a b:

$\mathrm{Lab}_{out}(e) = \mathrm{Lab}_{in}(b) \wedge$

Lab⇩i⇩n(b) $= \mathrm{Lab}_{in}(e)+1 \wedge$

Env⇩i⇩n(a) $= \mathrm{Env}_{in}(e) \wedge$

Env⇩i⇩n(b) $= \mathrm{Env}_{in}(e) \wedge$

$\mathrm{Code}_{out}(e) = \mathrm{Code}_{out}\ (a)$
@ [⟨CMP,r7,#0⟩]
@ [⟨BNQ L $\mathrm{Lab}_{in}(e)+1$⟩]
@ $\mathrm{Code}_{out}\ (b)$
@ [⟨L $\mathrm{Lab}_{in}(e) + 1$⟩]

Optimising Code Generation

Optimising Code Generation

- Optimisations are done on various levels, e.g.

Optimising Code Generation

- Optimisations are done on various levels, e.g.
  - Source code level:
    (challenge : data-flow analysis,
     advantage: type information available)

Optimising Code Generation

- Optimisations are done on various levels, e.g.
  - Source code level:
    (challenge : data-flow analysis,
    advantage: type information available)
  - Assembler code level:
    (advantage: control flow simple,
    sometimes type-info hampers view
    on machine-level executions and their
    optimisation potential)

Optimising Code Generation

- Optimisations are done on various levels, e.g.
  - Source code level:
    (challenge : data-flow analysis,
    advantage: type information available)
  - Assembler code level:
    (advantage: control flow simple,
    sometimes type-info hampers view
    on machine-level executions and their
    optimisation potential)

Optimising Code Generation

Optimising Code Generation

- For sake of demonstration, we do it low-level, by transforming patterns of assembly code:

Optimising Code Generation

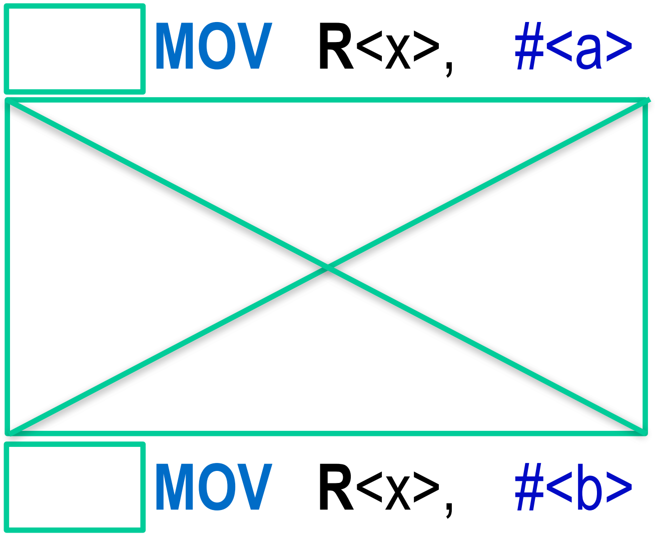
- For sake of demonstration, we do it low-level, by transforming patterns of assembly code:
  - Constant Propagation (as “peephole optim.”)

Optimising Code Generation

- For sake of demonstration, we do it low-level, by transforming patterns of assembly code:
  - Constant Propagation (as “peephole optim.”)

Optimising Code Generation

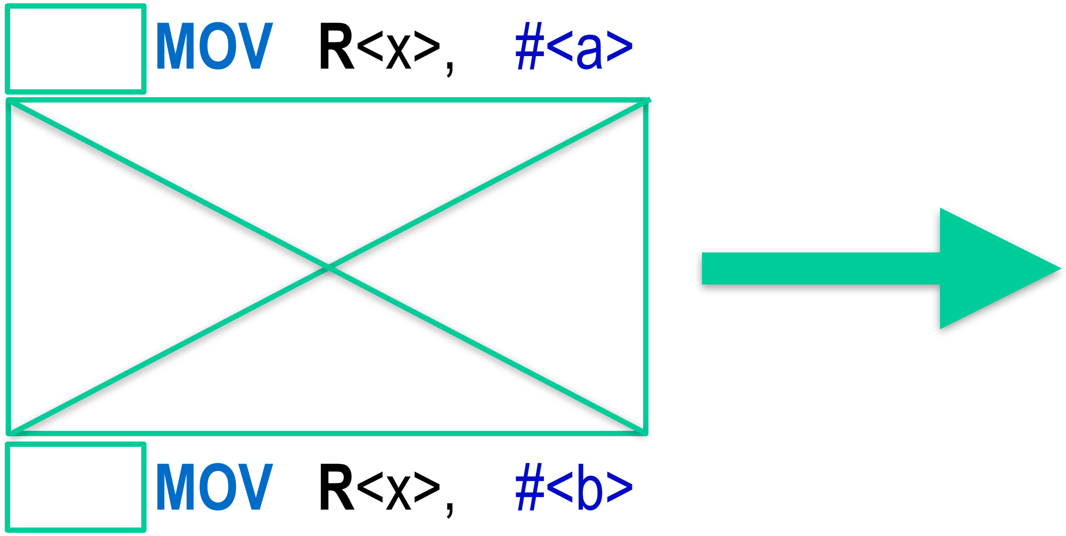
- For sake of demonstration, we do it low-level, by transforming patterns of assembly code:
  - Constant Propagation (as "peephole optim.")



```
MOV  R<x>,  #<a>

MOV  R<x>,  #<b>
```

Optimising Code Generation

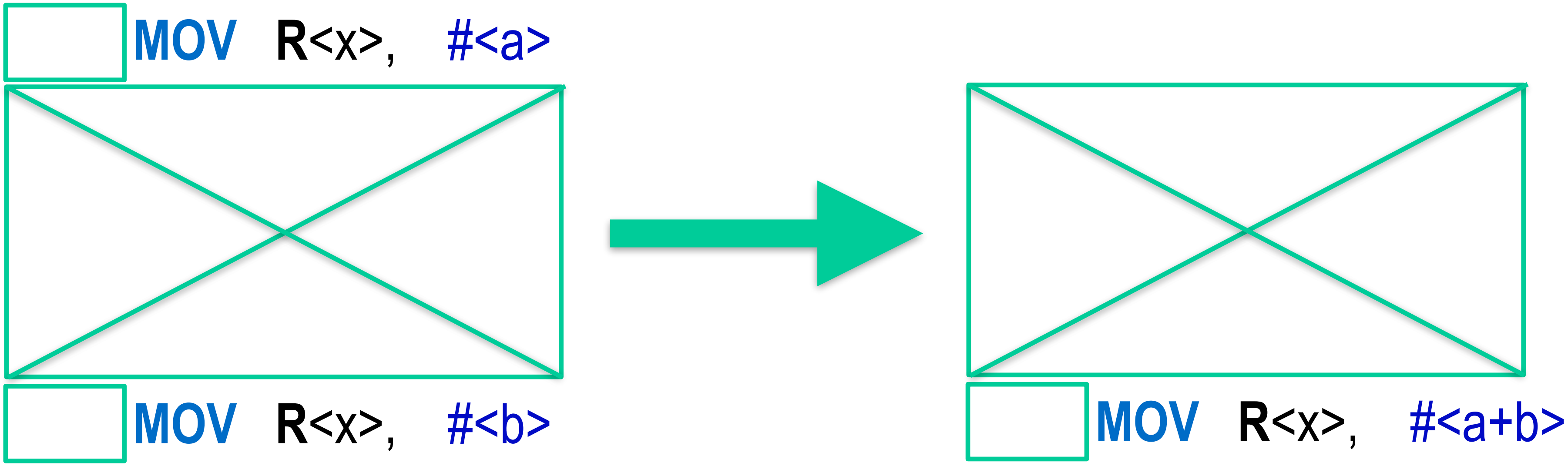
- For sake of demonstration, we do it low-level, by transforming patterns of assembly code:
  - Constant Propagation (as “peephole optim.”)



```
MOV  R<x>,  #<a>
MOV  R<x>,  #<b>
```

Optimising Code Generation

- For sake of demonstration, we do it low-level, by transforming patterns of assembly code:
  - Constant Propagation (as "peephole optim.")

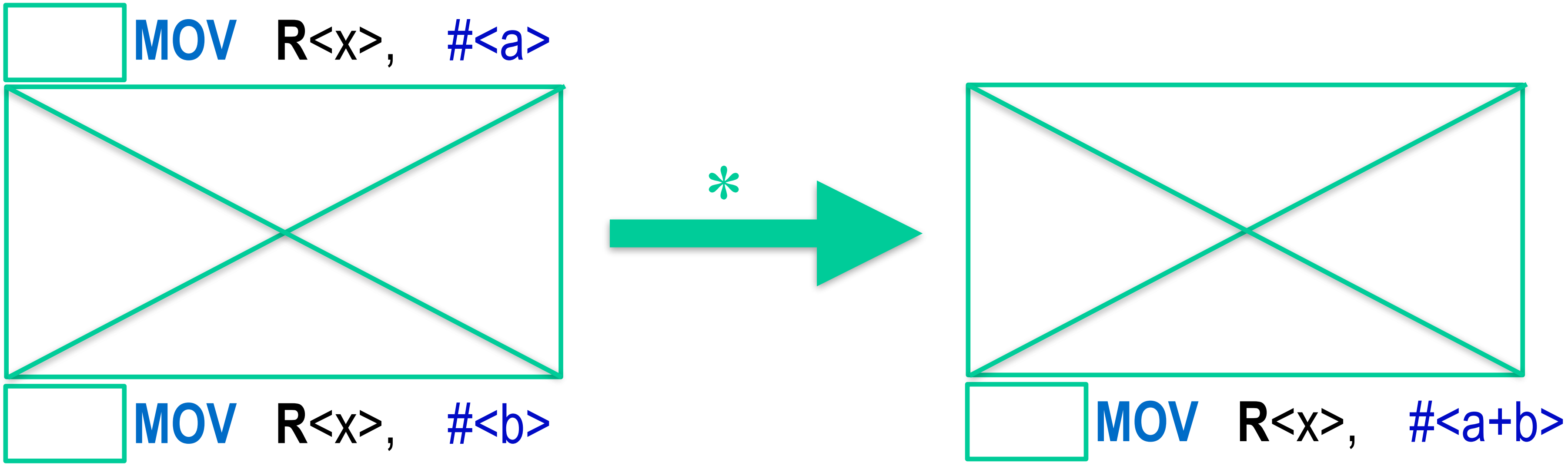


```
MOV  R<x>,  #<a>

MOV  R<x>,  #<b>
```

```
MOV  R<x>,  #<a+b>
```

Optimising Code Generation

- For sake of demonstration, we do it low-level, by transforming patterns of assembly code:
  - Constant Propagation (as "peephole optim.")



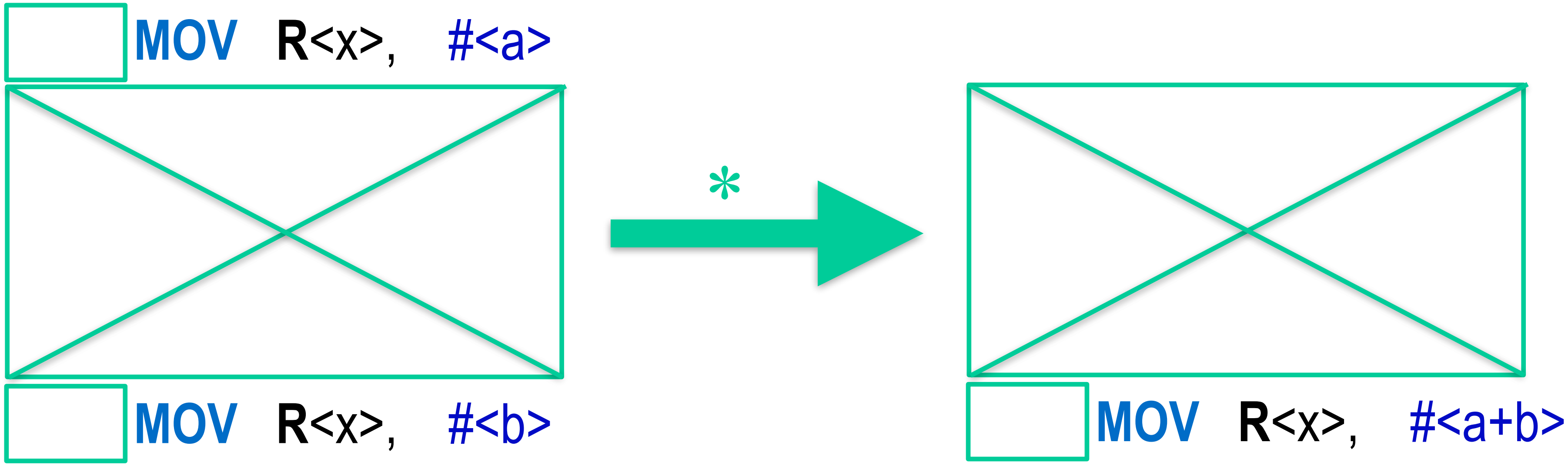
```
MOV  R<x>,  #<a>

MOV  R<x>,  #<b>
```

*

```
MOV  R<x>,  #<a+b>
```

Optimising Code Generation

- For sake of demonstration, we do it low-level, by transforming patterns of assembly code:
  - Constant Propagation (as “peephole optim.”)



```
MOV  R<x>,  #<a>

MOV  R<x>,  #<b>
```

```
MOV  R<x>,  #<a+b>
```

* no label, no ref to R<x>, a+b < 2^16

Optimising Code Generation

Optimising Code Generation

- For sake of demonstration, we do it low-level, by transforming patterns of assembly code:

Optimising Code Generation

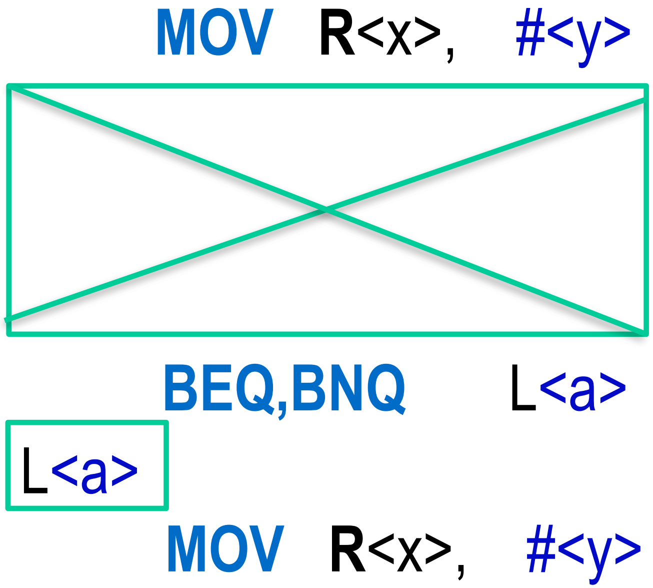
- For sake of demonstration, we do it low-level, by transforming patterns of assembly code:
  - Branch Propagation (as "peephole optim.")

Optimising Code Generation

- For sake of demonstration, we do it low-level, by transforming patterns of assembly code:
  - Branch Propagation (as “peephole optim.”)

Optimising Code Generation

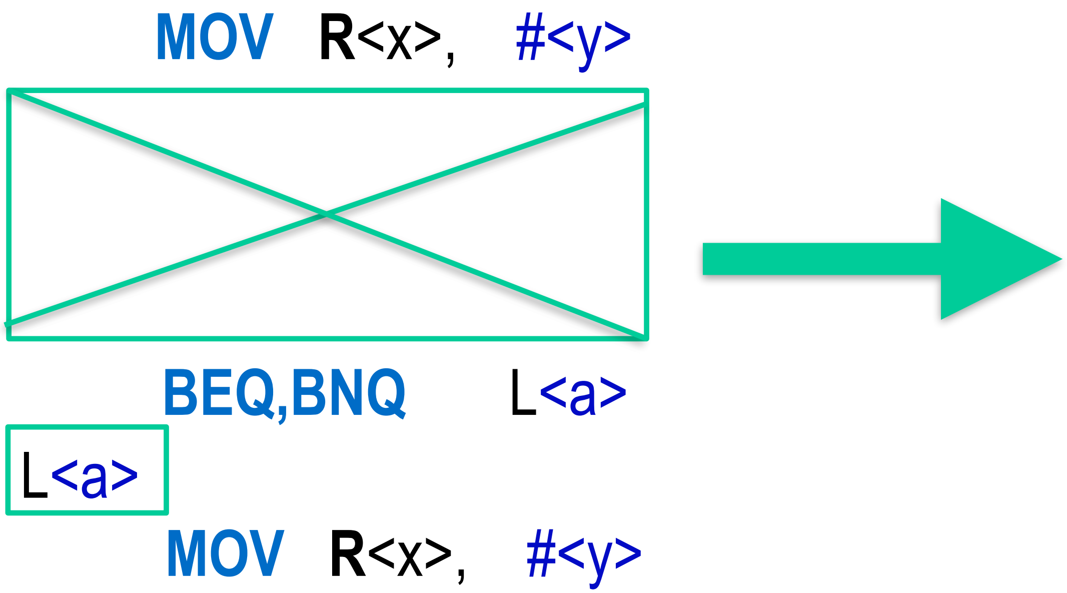
- For sake of demonstration, we do it low-level, by transforming patterns of assembly code:
  - Branch Propagation (as “peephole optim.”)



```
        MOV   R<x>,   #<y>

        BEQ,BNQ      L<a>
L<a>
        MOV   R<x>,   #<y>
```

Optimising Code Generation

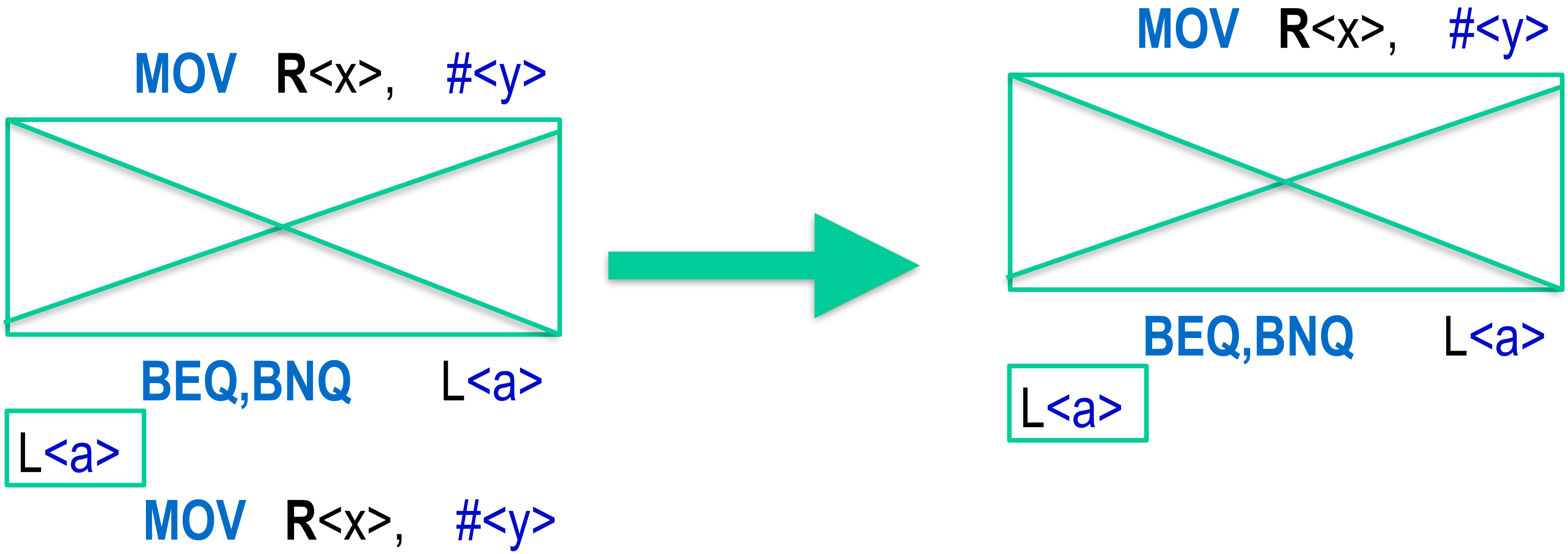
- For sake of demonstration, we do it low-level, by transforming patterns of assembly code:
  - Branch Propagation (as "peephole optim.")



```
      MOV   R<x>,   #<y>

      BEQ,BNQ     L<a>
L<a>
      MOV   R<x>,   #<y>
```

Optimising Code Generation

- For sake of demonstration, we do it low-level, by transforming patterns of assembly code:
  - Branch Propagation (as "peephole optim.")



MOV R<x>, #<y>

BEQ,BNQ L<a>

L<a>

MOV R<x>, #<y>

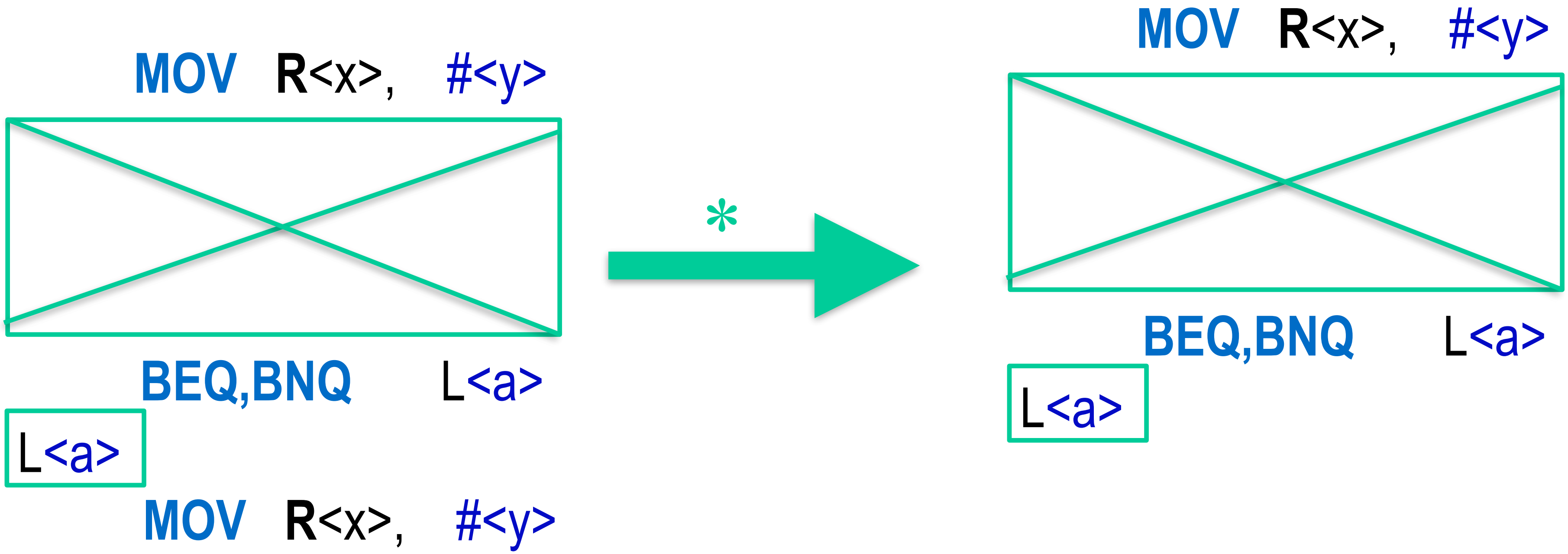
MOV R<x>, #<y>

BEQ,BNQ L<a>

L<a>

Optimising Code Generation

- For sake of demonstration, we do it low-level, by transforming patterns of assembly code:
  - Branch Propagation (as "peephole optim.")



MOV R<x>, #<y>

BEQ,BNQ L<a>

L<a>

MOV R<x>, #<y>

*

MOV R<x>, #<y>

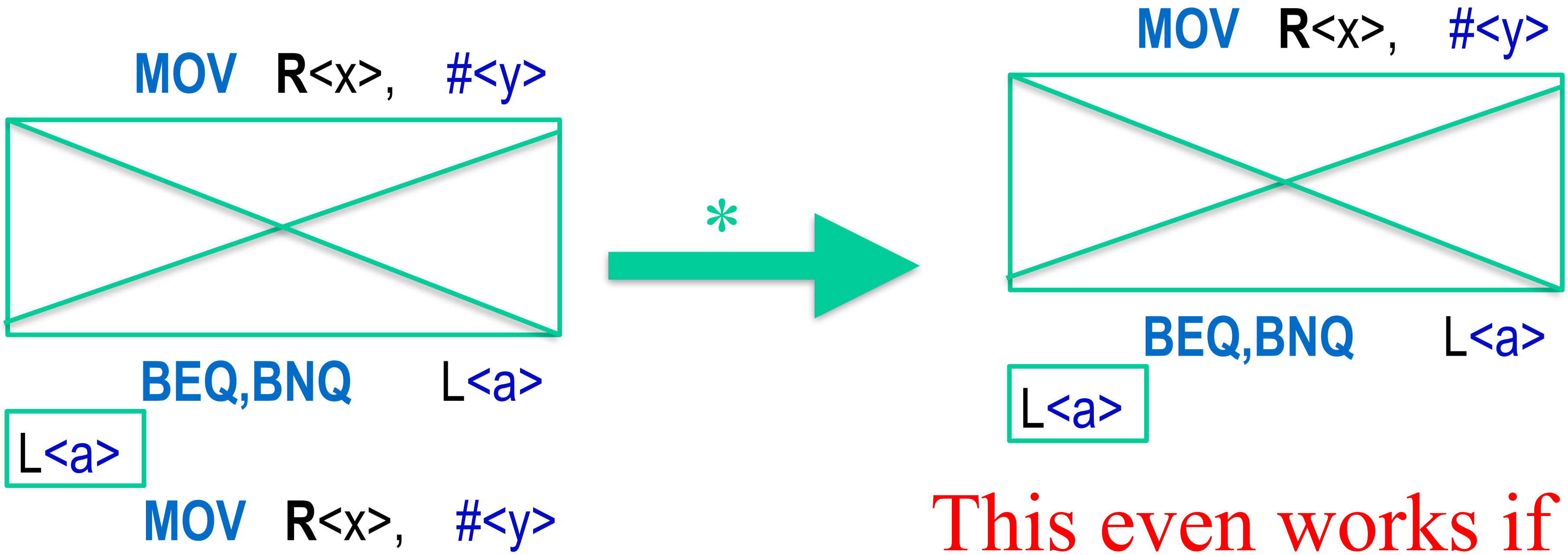
BEQ,BNQ L<a>

L<a>

* no label, no ref to R<x>

Optimising Code Generation

- For sake of demonstration, we do it low-level, by transforming patterns of assembly code:
  - Branch Propagation (as "peephole optim.")



MOV R<x>, #<y>

BEQ,BNQ L<a>

L<a>

MOV R<x>, #<y>

*

MOV R<x>, #<y>

BEQ,BNQ L<a>

L<a>

* no label, no ref to R<x>

This even works if the types are different in the source language !

Optimising Code Generation

```
MUL  R<x>, R<y>, #0x2          ADD  R<x>, R<y>, R<y>
```

Optimising Code Generation

- For sake of demonstration, we do it low-level, by transforming patterns of assembly code:

```
MUL  R<x>, R<y>, #0x2                    ADD  R<x>, R<y>, R<y>
```

Optimising Code Generation

- For sake of demonstration, we do it low-level, by transforming patterns of assembly code:
  - Classic: Arith ops optimisation:

```
MUL  R<x>, R<y>, #0x2                    ADD  R<x>, R<y>, R<y>
```

Optimising Code Generation

- For sake of demonstration, we do it low-level, by transforming patterns of assembly code:
  - Classic: Arith ops optimisation:

```
MUL  R<x>, R<y>, #0x2                    ADD  R<x>, R<y>, R<y>
```

Optimising Code Generation

- For sake of demonstration, we do it low-level, by transforming patterns of assembly code:
  - Classic: Arith ops optimisation:

```
MUL  R<x>, R<y>, #0x2    →    ADD  R<x>, R<y>, R<y>
```

Optimising Code Generation

- For sake of demonstration, we do it low-level, by transforming patterns of assembly code:
  - Classic: Arith ops optimisation:

MUL R<x>, R<y>, #0x2 $\xrightarrow{*}$ **ADD** R<x>, R<y>, R<y>

Optimising Code Generation

- For sake of demonstration, we do it low-level, by transforming patterns of assembly code:
  - Classic: Arith ops optimisation:

MUL **R**<x>, **R**<y>, #0x2 $\xrightarrow{*}$ **ADD** **R**<x>, **R**<y>, **R**<y>

In some architectures, this is mapped to SHL (shift-left)

Conclusion

Conclusion

- Code generation is - by definition - dependent of the target architecture

Conclusion

- Code generation is - by definition - dependent of the target architecture
- … and its core, the ISA

Conclusion

- Code generation is - by definition - dependent of the target architecture
- … and its core, the ISA
- Modern ISA's are X86, ARM, RISC V, and virtual ISA's like JVM or LLVM

Conclusion

- Code generation is - by definition - dependent of the target architecture
- … and its core, the ISA
- Modern ISA's are X86, ARM, RISC V, and virtual ISA's like JVM or LLVM
- There are numerous open-source tools for ISA's and the corresponding compilers …

Conclusion

- Code generation is - by definition - dependent of the target architecture
- ... and its core, the ISA
- Modern ISA's are X86, ARM, RISC V, and virtual ISA's like JVM or LLVM
- There are numerous open-source tools for ISA's and the corresponding compilers ...

Conclusion

Conclusion

- Code-Generation can be specified as Syntax-Directed Translation

Conclusion

- Code-Generation can be specified as Syntax-Directed Translation
- … and also verifiable by proof techniques

Conclusion

- Code-Generation can be specified as Syntax-Directed Translation
- … and also verifiable by proof techniques
- Optimisations make modern compilers unbeatable in terms of target code efficiency !

Conclusion

- Code-Generation can be specified as Syntax-Directed Translation
- … and also verifiable by proof techniques
- Optimisations make modern compilers unbeatable in terms of target code efficiency !

References

Tutorial Cortex M7 https://www.youtube.com/watch?v=JmpQ79h_0eA&list=PL3hGN82ZNBxi111hEdE0VhGVP_jMhN9cO&index=5

https://www.cl.cam.ac.uk/events/cl75/posters/f/acjf3-screen.pdf

https://godbolt.org/