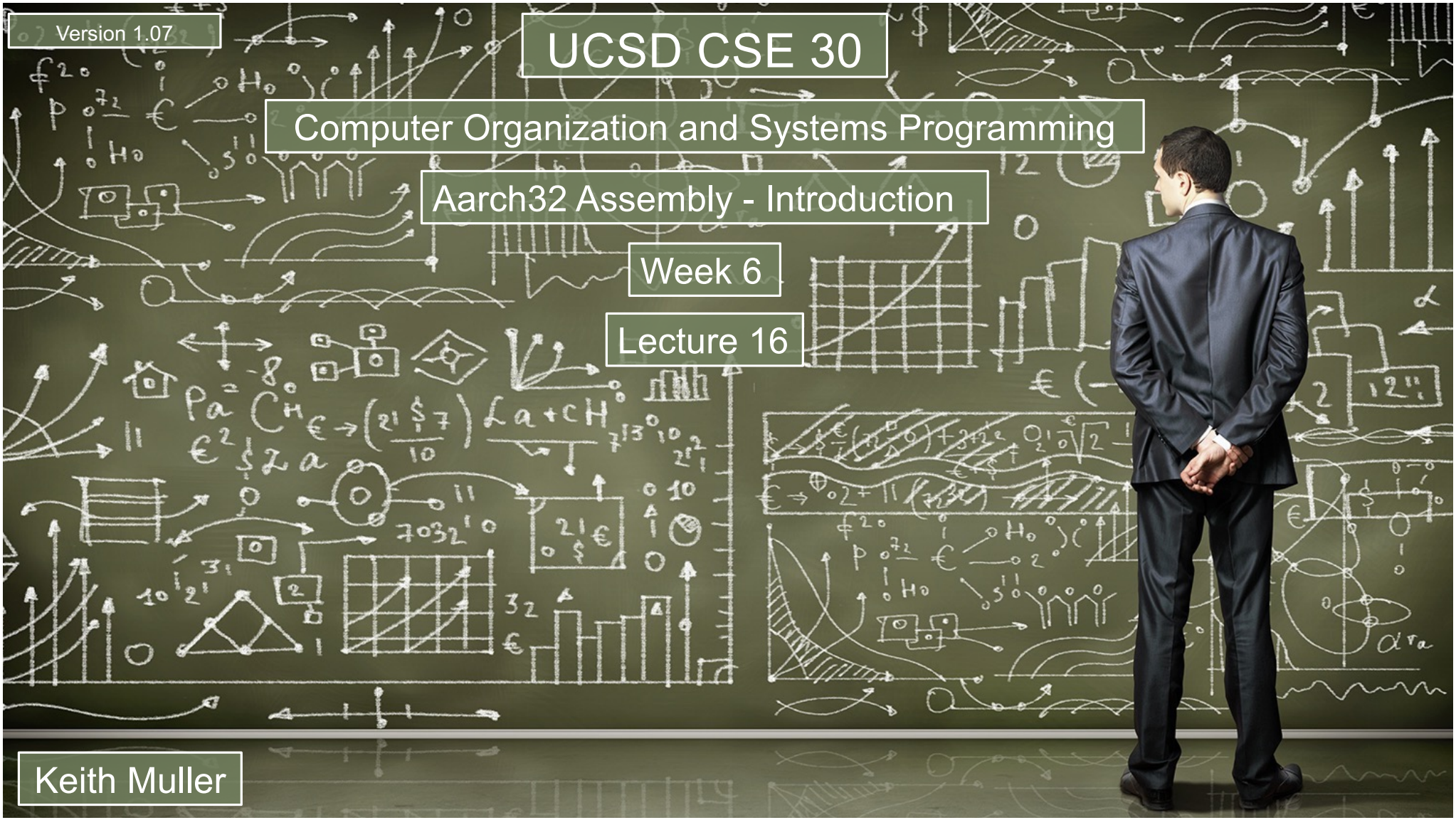
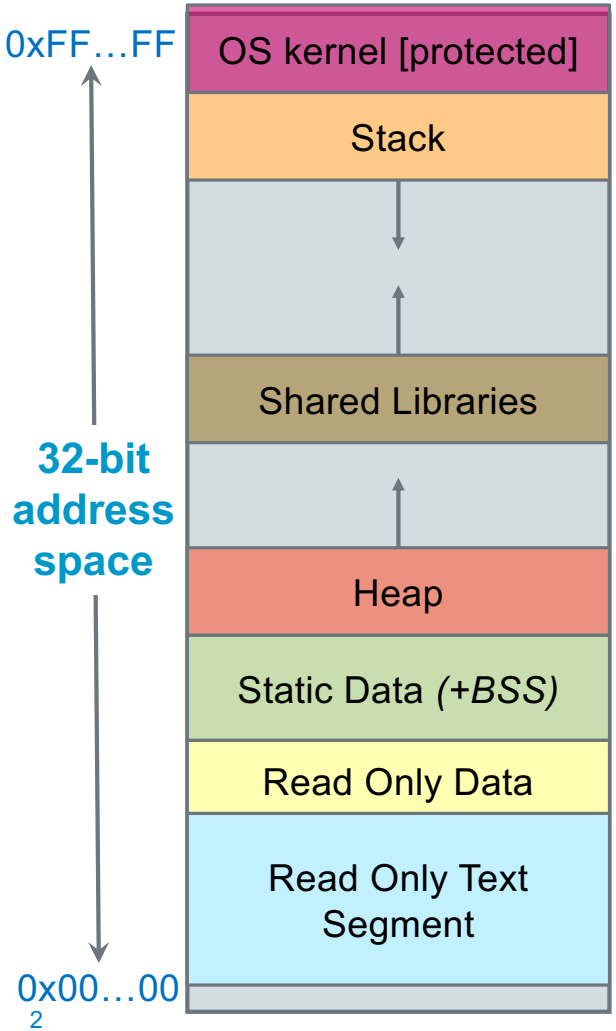

Version 1.07
UCSD CSE 30
Computer Organization and Systems Programming
Aarch32 Assembly - Introduction
Week 6
Lecture 16
Keith Muller

Assembly and Machine Code



- Machine Language (or code): Set of instructions the CPU executes are encoded in memory using patterns of ones and zeros
- Assembly language is a symbolic version of the machine language
- Each assembly statement (called an **Instruction**), executes exactly **one** from a list of simple commands
  - Instructions describe operations (e.g., =, +, -, *)
- Each line of arm32 assembly code contains at most one instruction
- Assembler (gnu as) translates assembly to machine code

Memory Address	word (4-bytes) contents	Assembly Language
1040c:	e28db004	add fp, sp, 4
10410:	e59f0010	ldr r0, [pc, 16]
10414:	ebffffb3	bl 102e8 <printf>
10418:	e3a00000	mov r0, 0
1041c:	e24bd004	sub sp, fp, 4

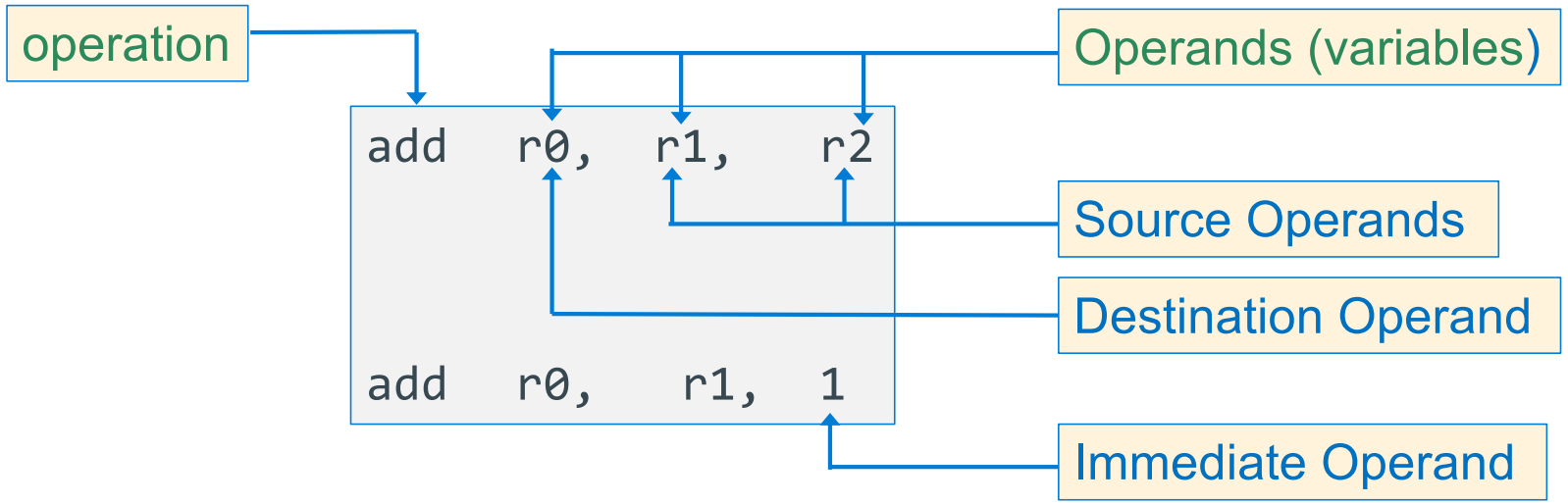
Machine Code

high <- low bytes

x

Anatomy of an Assembly instruction

- Assembly language instructions specify an operation and the operands to the instruction (arguments of the operation)
- Three basic types of operands
  - Destination: where the result will be stored
  - Source: where data is read from
  - Immediate: an actual value like the 1 in y = x + 1



x

Meaning of an Instruction

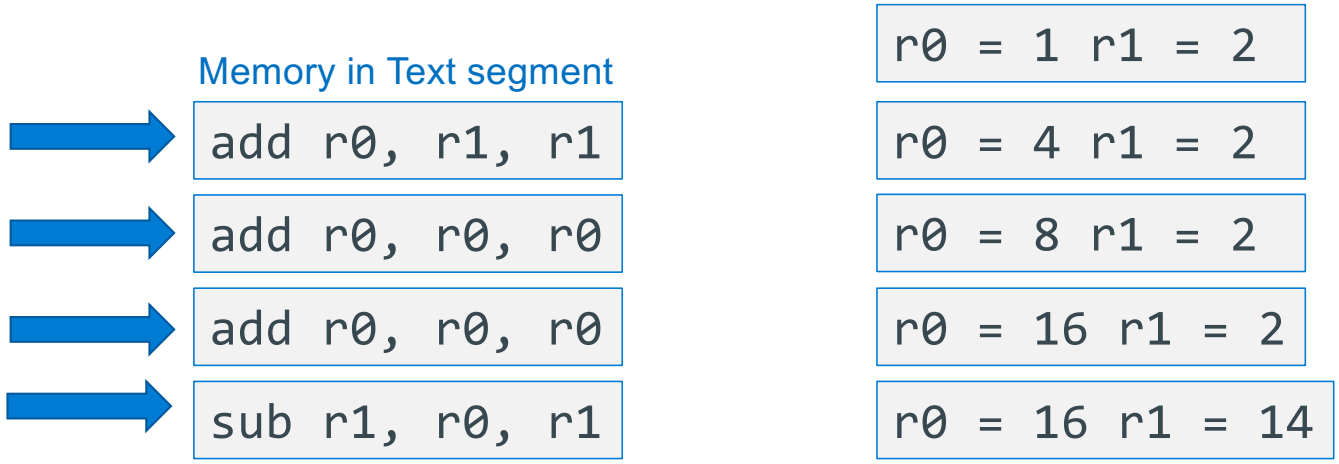
- Operations are abbreviated int opcodes (1– 5 letters)
- Assembly Instructions are specified with a very regular syntax
  - Opcodes are followed by arguments
  - Usually the destination argument is next, then one or more source arguments (this is not strictly the case, but it is generally true)
- Why this order?
- Analogy to C or Java

```
int r0, r1, r2;
    r0 = r1 + r2;

       /*  r0 = r1 +  r2 */
    add    r0,  r1,   r2
```


x

Program Execution: A Series of Instructions

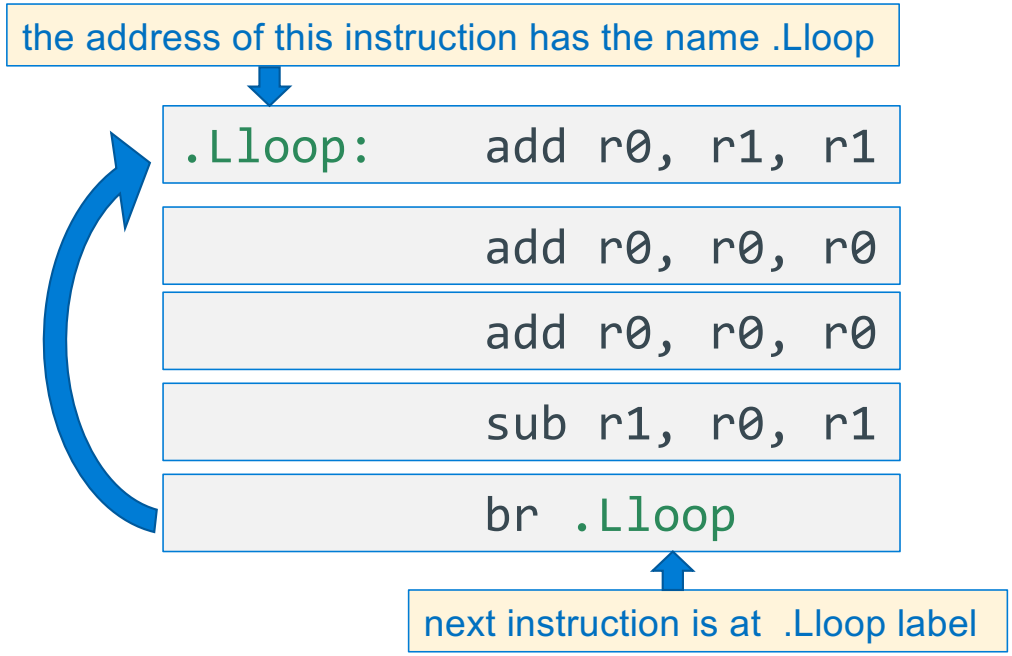
- Generally (there are exceptions like function calls, loops etc.)
- Instructions are retrieved sequentially from memory
- An instruction executes to completion before the next instruction is completed
- But there are exceptions to this (later and in cse 141, cse 142)



x

Program Execution: Looping in the Execution Flow

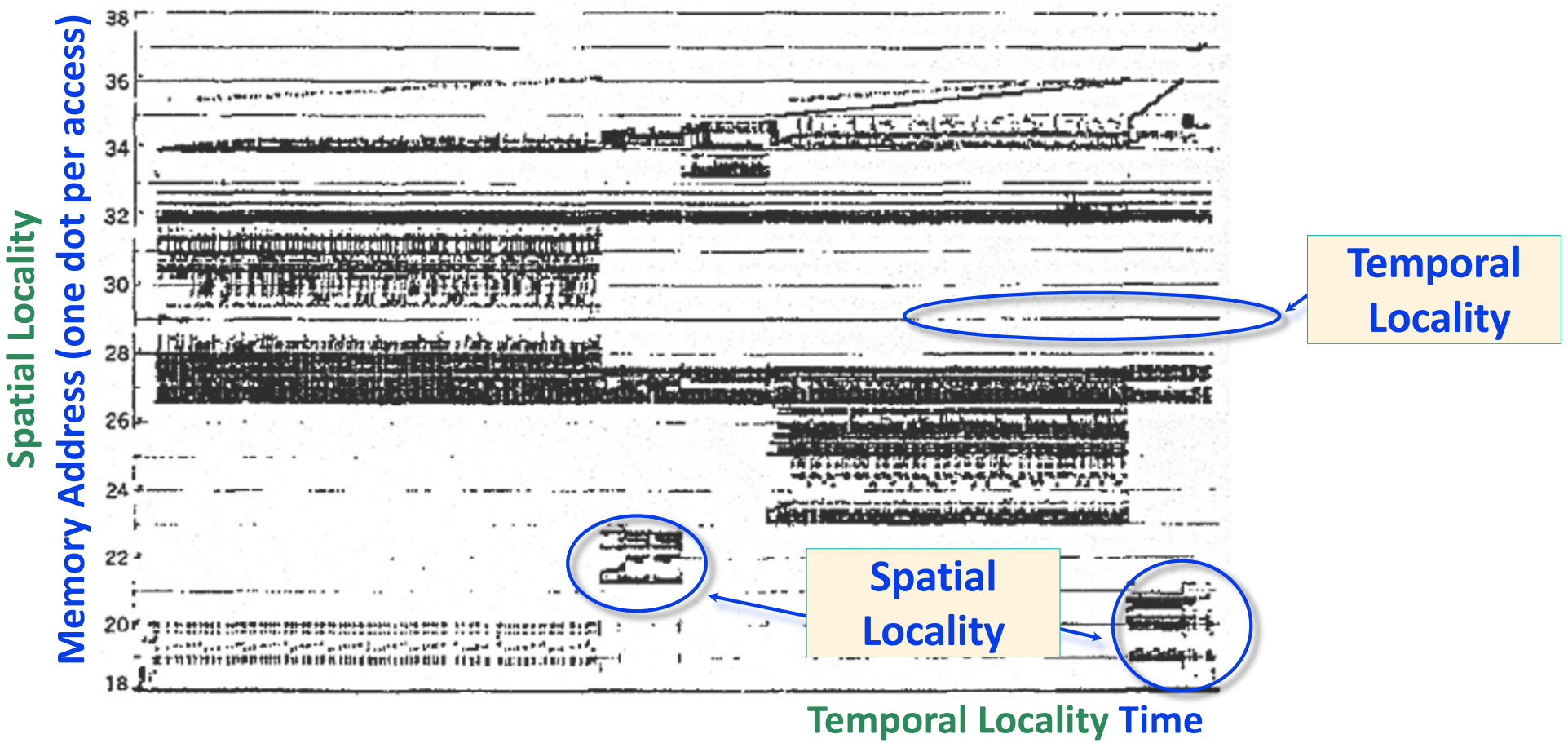
- Repeat the series of instructions in a loop means altering the flow of execution
- This is used with if statements and loops
- Below is an example of an infinite loop




```
r0 = 1 r1 = 2
r0 = 4 r1 = 2
r0 = 8 r1 = 2
r0 = 16 r1 = 2
r0 = 16 r1 = 14
```

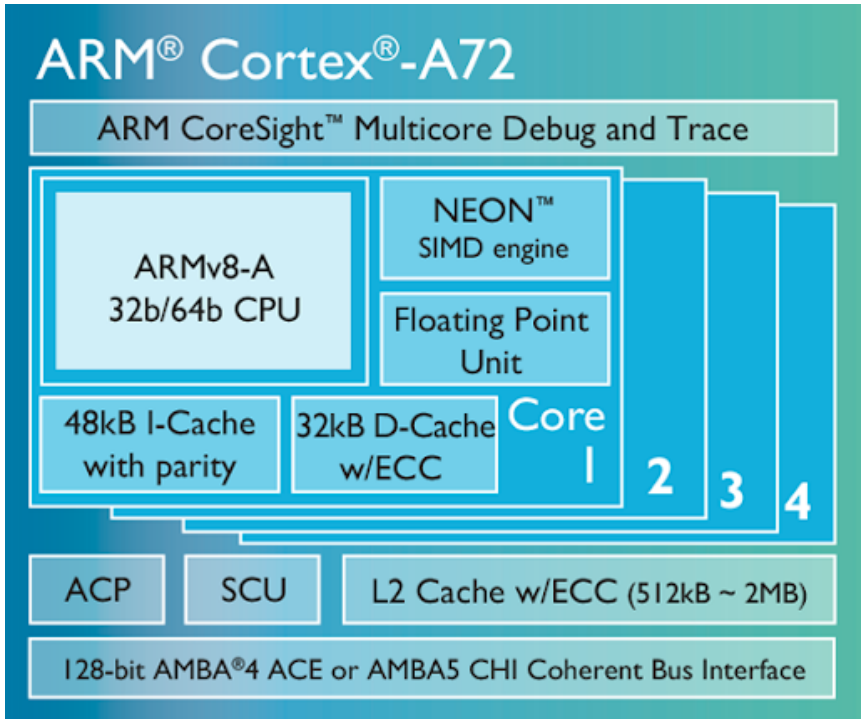

x

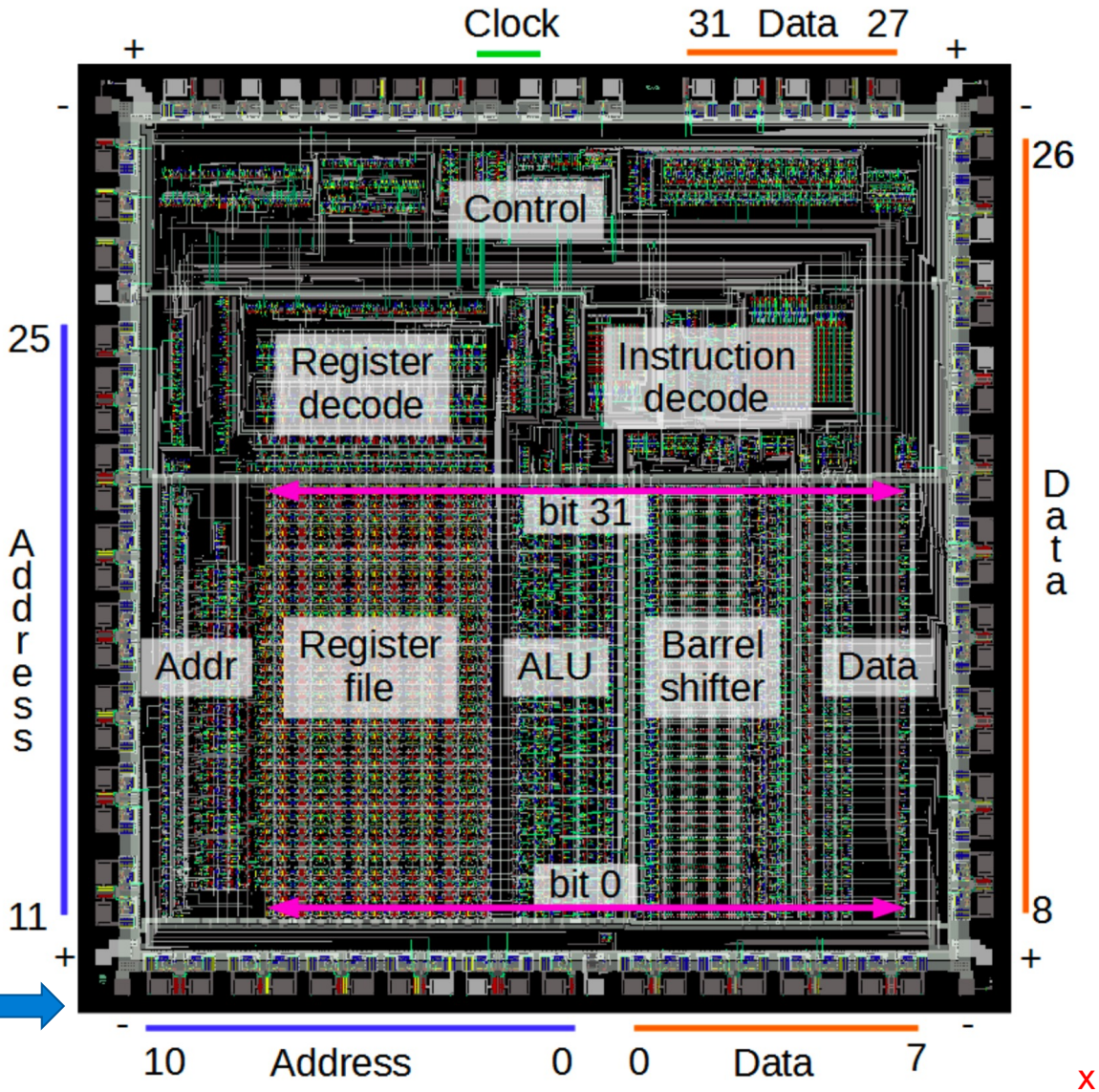
Principle of Locality: Spatial and Temporal Memory Access of Executing Programs



Donald J. Hatfield, Jeanette Gerald: Program Restructuring for Virtual Memory. IBM Systems Journal 10(3): 168-192 (1971)

Arm Core Organization & Floorplan Examples





Single core ***arm*** die (not an A72) ***Floorplan***

Version 1.07

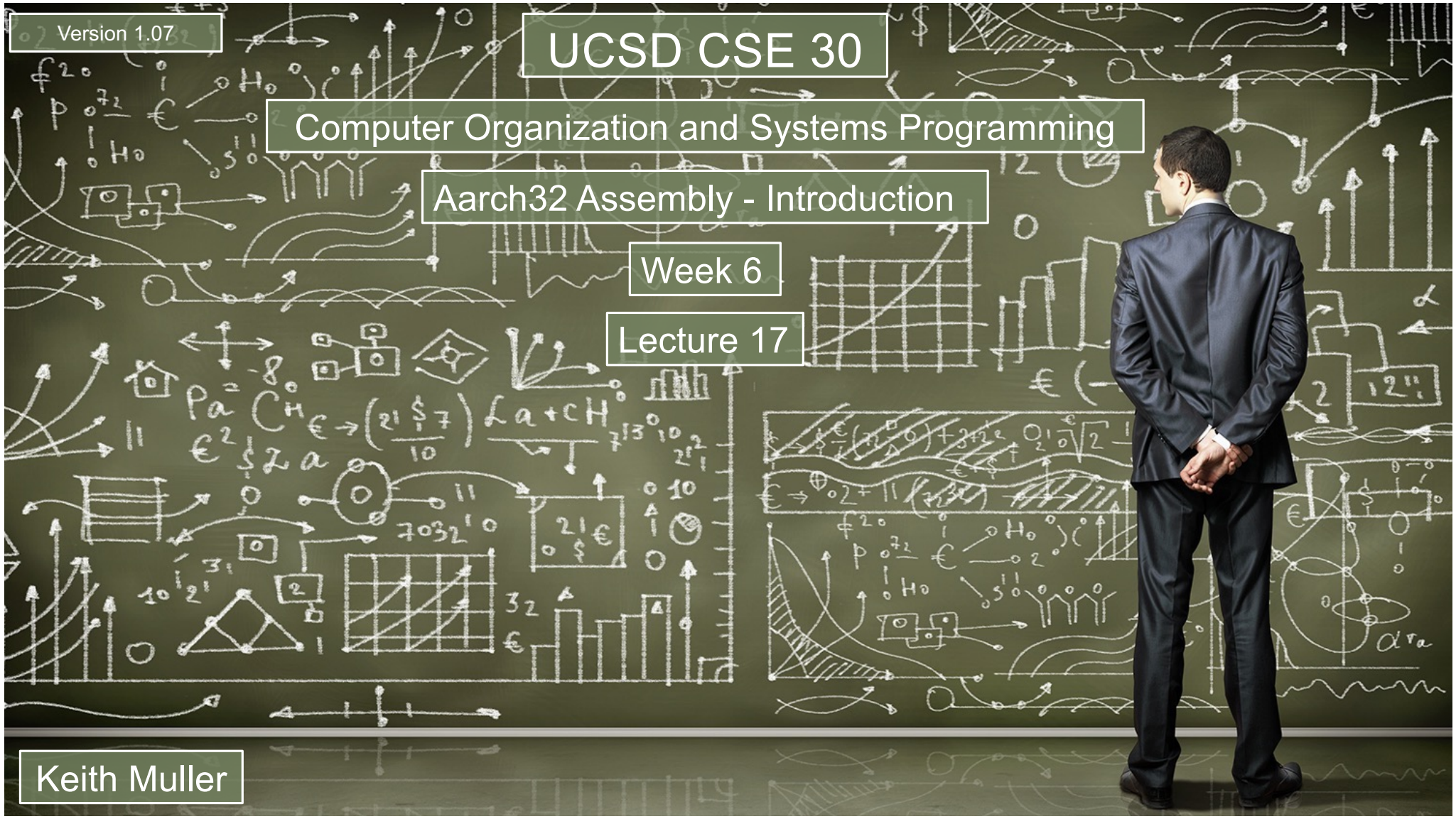
UCSD CSE 30

Computer Organization and Systems Programming

Aarch32 Assembly - Introduction

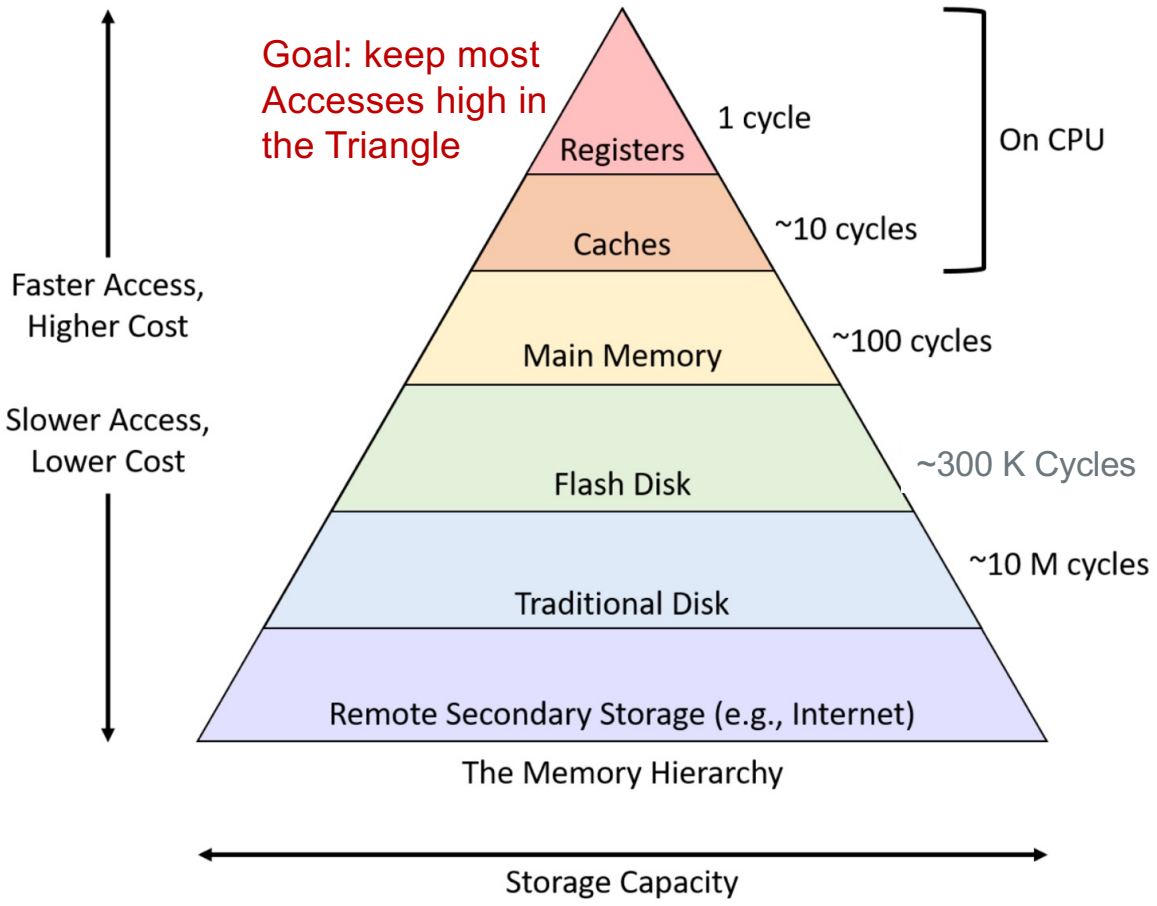
Week 6

Lecture 17

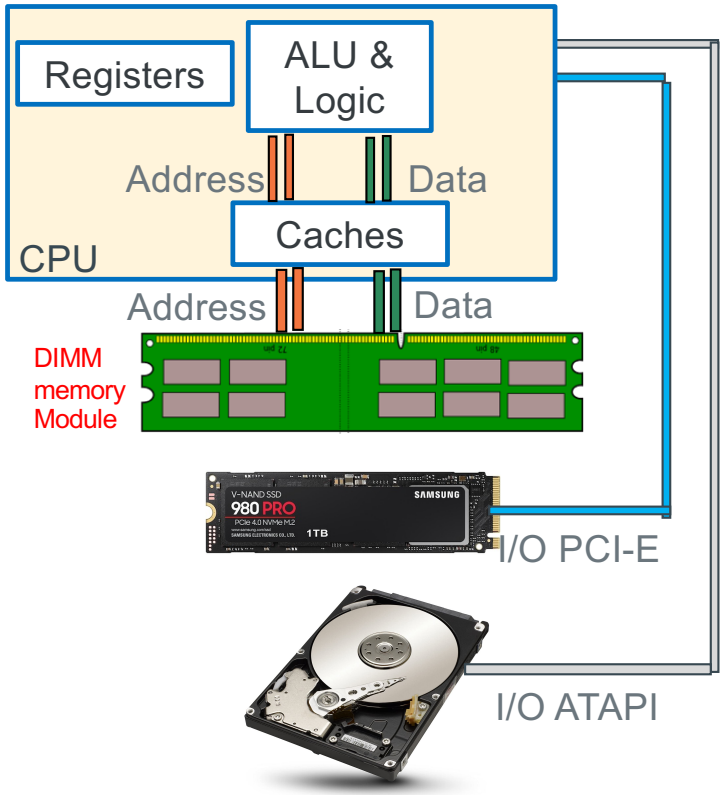


Keith Muller

Memory Triangle: Taking Advantage of Locality for Cost/Performance



Clock cycle =~ time to access; larger is slower

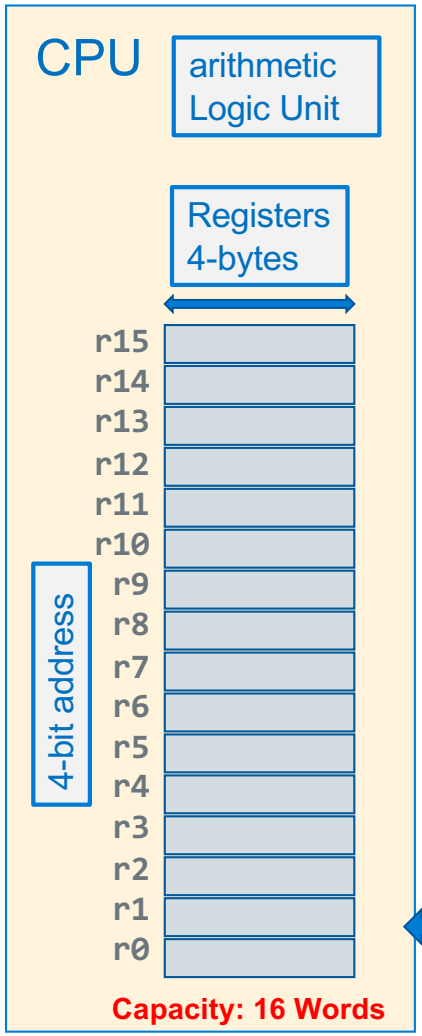


Assume 1 clock cycle per machine instruction

Source: Dive Into Systems Text

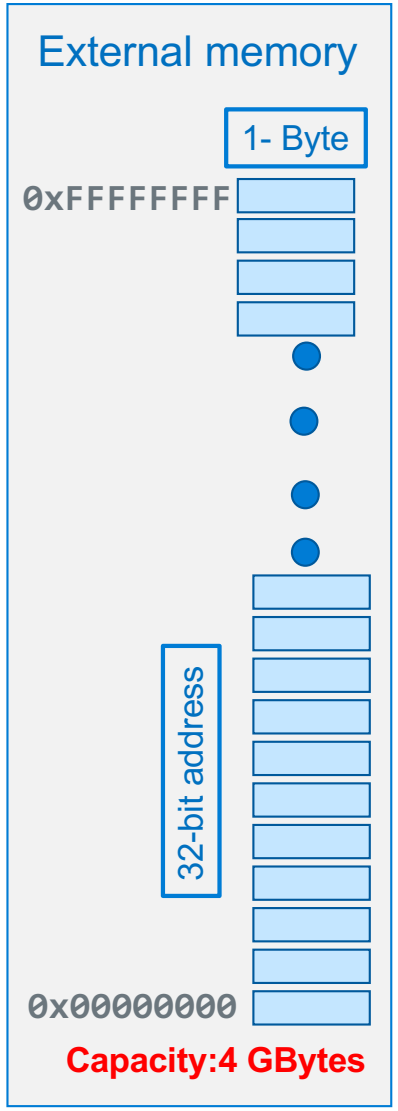
x

32-Bit Arm - Registers



- Registers are **memory located within the CPU**
- **Registers are the fastest** read and write storage
- Register is **word size in length** stores 32-bit values
  - Memory is accessed using pointers in registers
- In assembly language the registers have predefined names starting with an r to differentiate them from memory addresses which are labels (address)
- 16 registers: from r0 to r15 (encoded: 0x0 – 0xf)

CPU Memory Bus = Address + Data

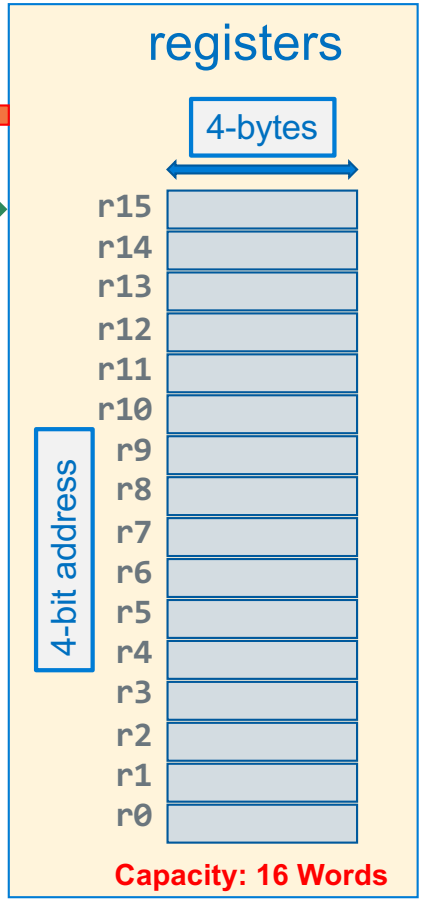


x

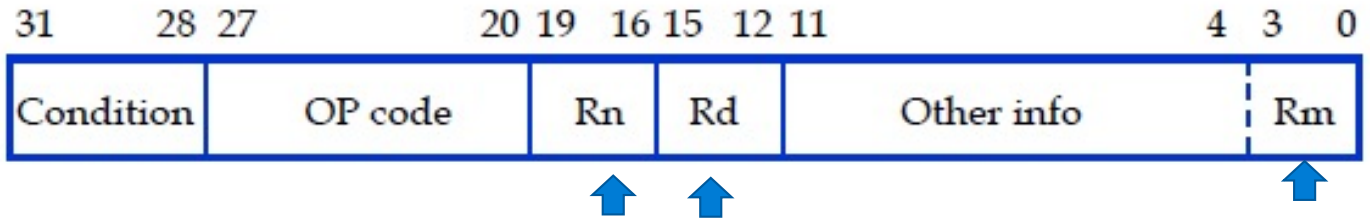
32-Bit Arm - Registers

All computations (add, subtract, etc.) are performed in the ALU

Arithmetic & Logic Unit (ALU)



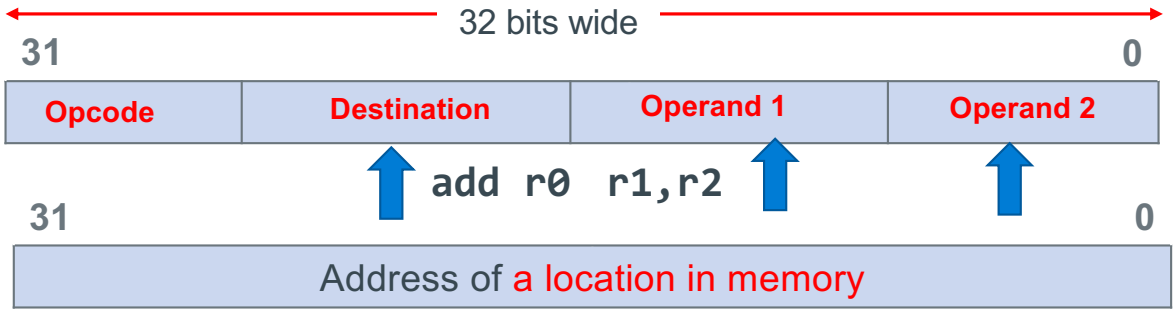
- Almost all arithmetic, logic operations and data movement operations involve at least one register
- As a result, Register addresses are **directly** encoded into 4-bit fields in machine instructions (see below)



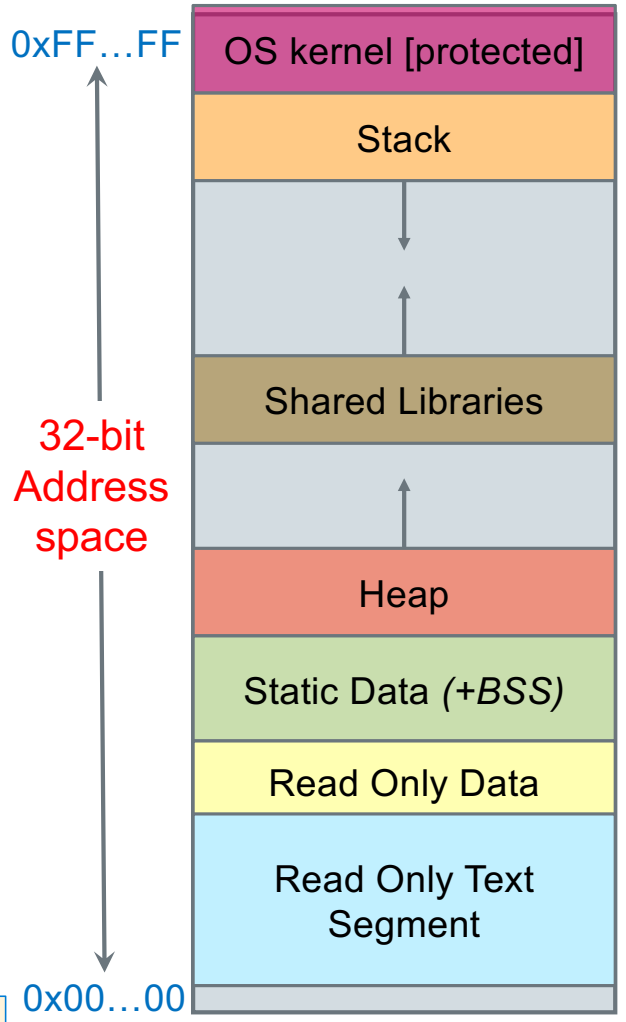
X

How to Access Memory?

- Consider r0 = r1 + r2
  - Operation code: add
  - Destination: r0
  - Operand 1: r1
  - Operand 2: r2
- Aarch32 Instructions are always word size: 32 bits wide
  - Some bits must be used to specify the operation code
  - Some bits must be used to specify the destination
  - Some bits must be used to specify the operands
- Address space is 32 bits wide – POINTERS in registers



NOT ENOUGH BITS for FULL Addresses in the instruction



x

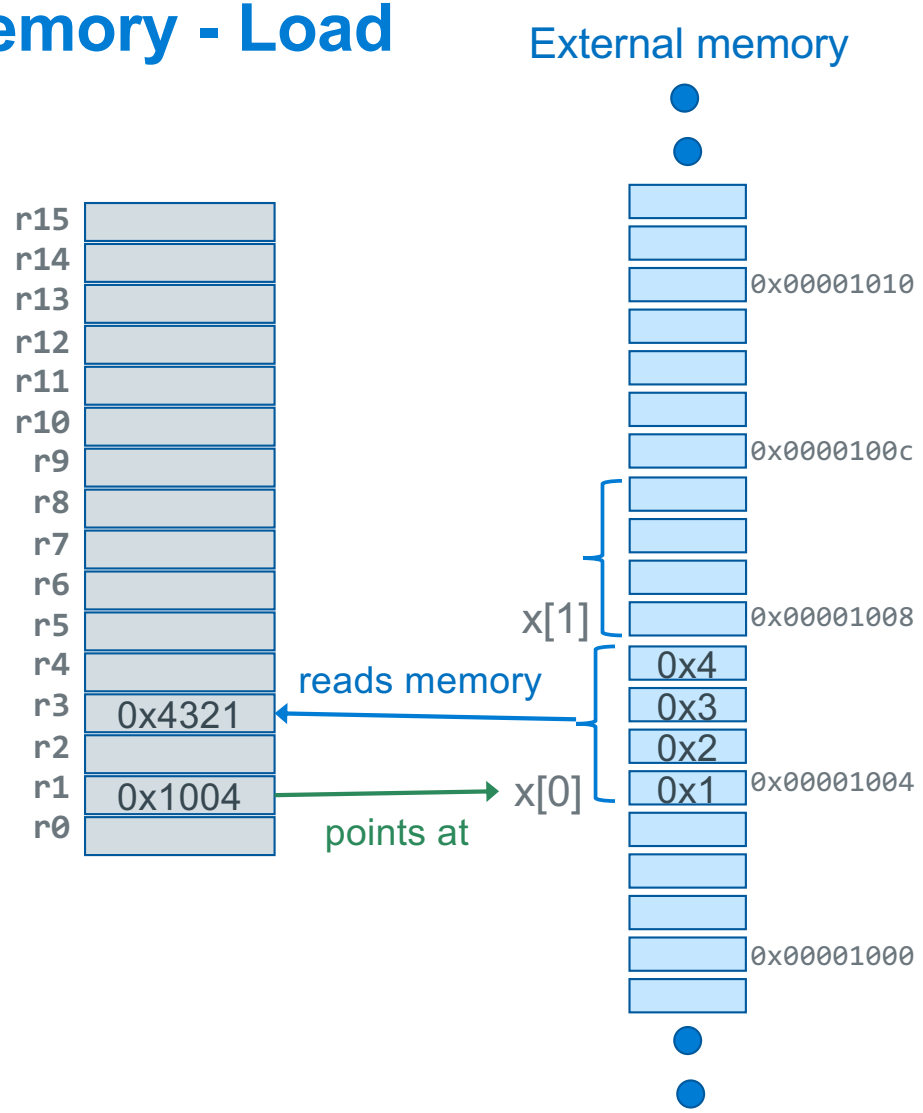
Using Registers as Pointers to Memory - Load

```
We want to do a x[1] = x[0]
We have to do this in two steps

int r3;
int x[2];
int *r1 = &x;     // r1 contains address
…
r3 = *(r1);       // memory to register
*(r1 + 1) = r3;   // register to memory
```

```
Load register from memory
ldr    r3, [r1, 0]
          ↑
address = r1 + 0 = 0x1004
```

The [ ] around the operands is like the * dereference op

we will cover this instruction in more detail later



x

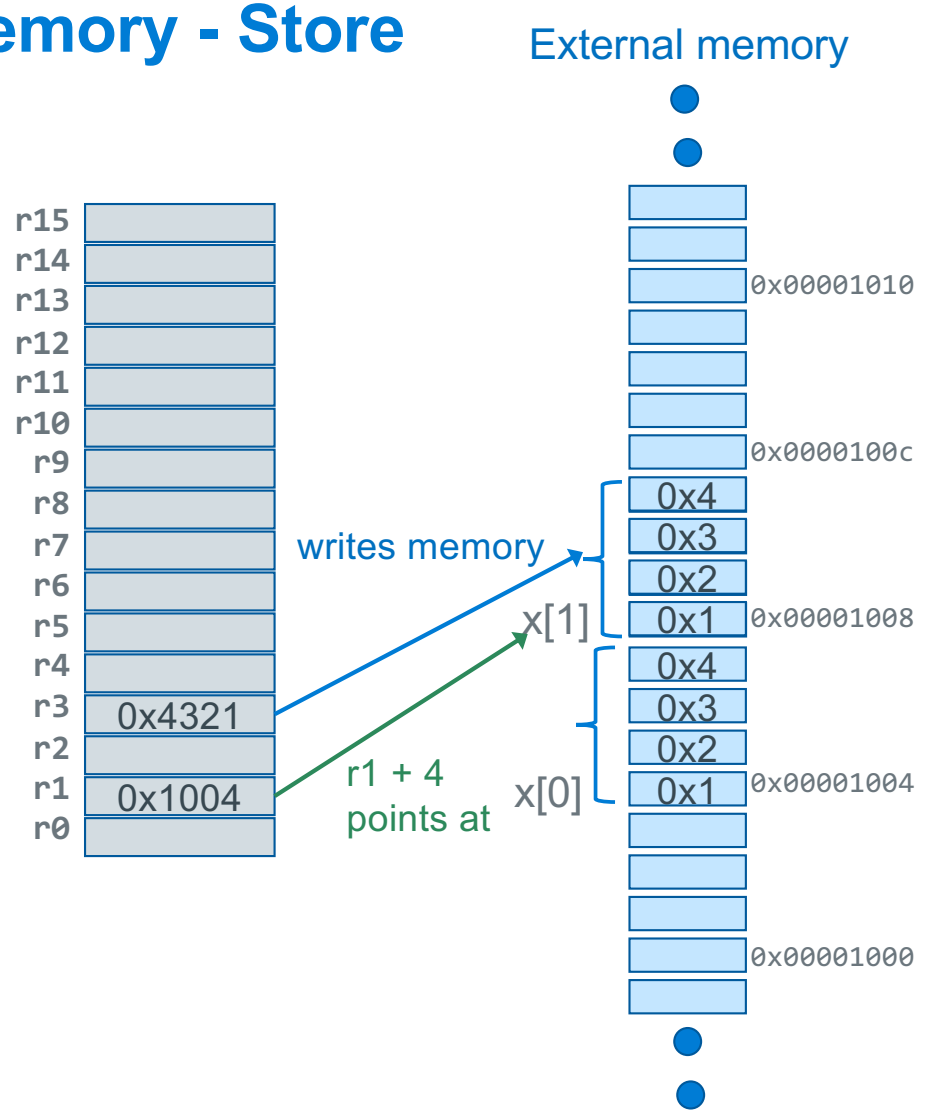
Using Registers as Pointers to Memory - Store

```
We want to do a x[1] = x[0]
We have to do this in two steps

int r3;
int x[2];
int *r1 = &x;     // r1 contains address
…
r3 = *(r1);       // memory to register
*(r1 + 1) = r3;   // register to memory
```

```
Store register to memory
str    r3, [r1, 4]

address = r1 + 4 = 0x1008
```

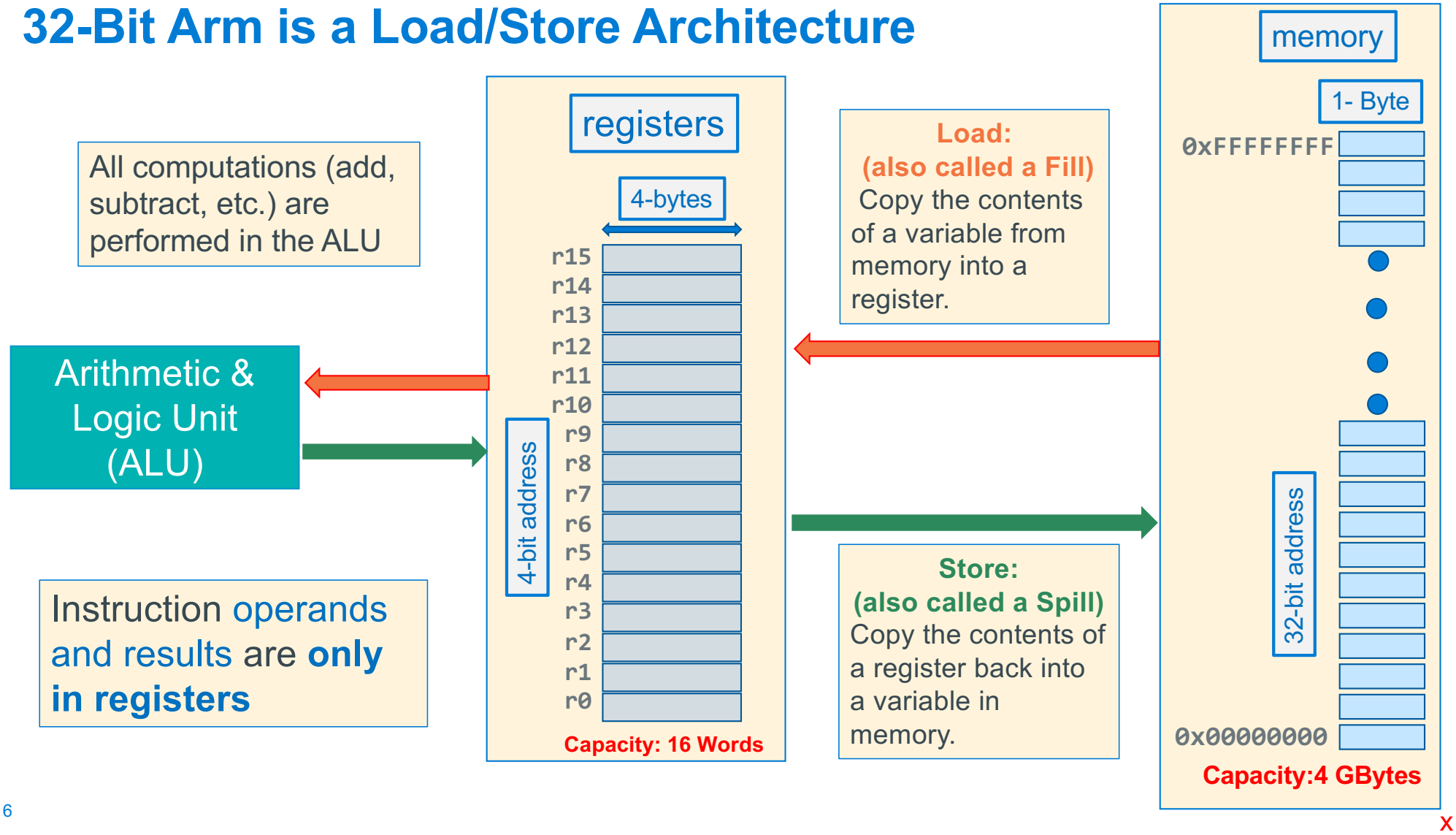
The [ ] around the operands is like the * dereference op

We will cover this instruction in more detail later



x

32-Bit Arm is a Load/Store Architecture



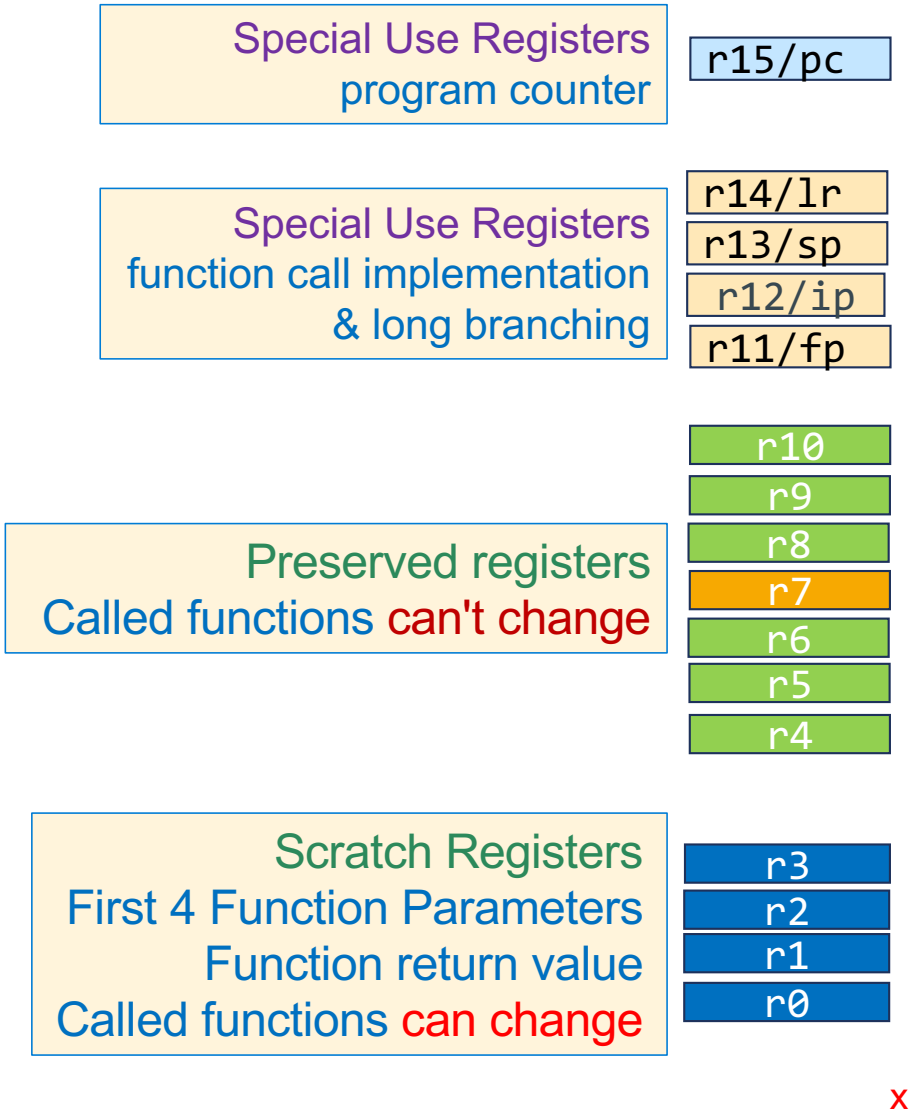
Arm Register Summary

- 16 Named registers r0 – r15
- The operands of almost all instructions are registers
- To operate on a variable in memory do the following:
  1. Load the value(s) from memory into a register
  2. Execute the instruction
  3. Store the result back into memory (only if needed!)
- Going to/from memory is expensive
  - 4X to 20X+ **slower** than accessing a register
- Strategy: Keep variables in registers as much as possible

x

Using Aarch32 Registers

- There are two basic groups of registers, general purpose and special use
- General purpose registers can be used to contain up to 32-bits of data, but you must follow the **rules** for their use
  - Rules specify how registers are to be used so software can communicate and share the use of registers (later slides)
- Special purpose registers: dedicated hardware use (like r15 the pc) or special use when used with certain instructions (like r13 & r14)
- r15/pc is the program counter that contains the address of an instruction being executed (not exactly … later)



x

How r1 (pc) is used: CPU Operational Overview

Everything has a **memory address**: **instructions** & **data**

- Machine code uses addresses for loops, branches, function calls, variables, etc.

1 Fetch

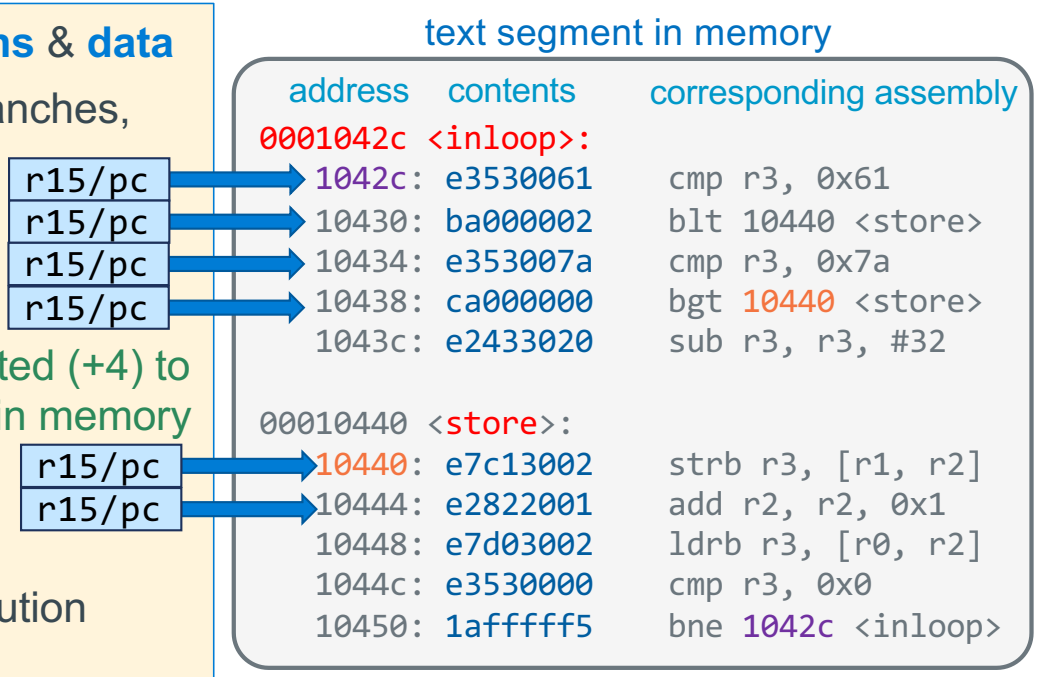
- read the instruction into memory ***(fetch)***
- program counter is automatically incremented (+4) to contain the address of the next instruction in memory
  Instructions are 32 bits

2 Decode

- ***Decodes the instruction*** and sets up execution

3 Execute

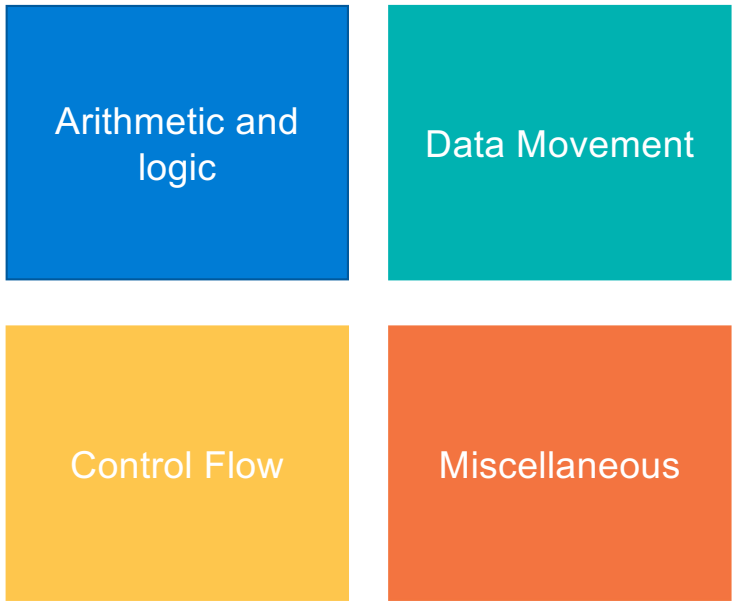
- CPU completes the ***execution*** of the instruction
- Execution may alter the pc to take branches, etc.
- Go to **fetch**



x

AArch32 Instruction Categories

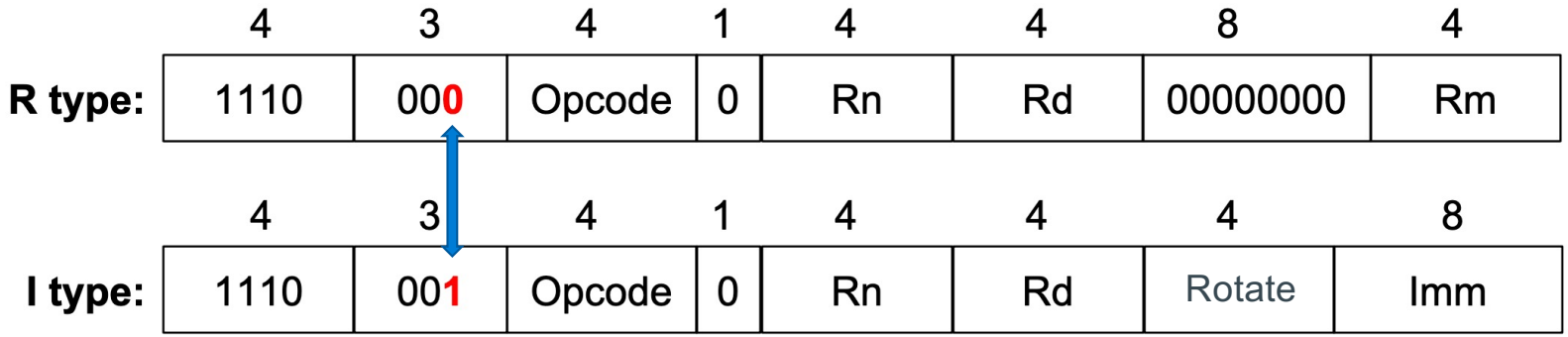
- **Data movement to/from memory**
  - Data Transfer Instructions between memory and registers
    - Load, Store
- **Arithmetic and logic**
  - Data processing Instructions (registers only)
    - Add, Subtract, Multiply, Shift, Rotate, …
- **Control Flow**
  - Compare, Test, If-then, Branch, function calls
- **Miscellaneous**
  - Traps (OS system calls), Breakpoints, wait for events, interrupt enable/disable, data memory barrier, data synchronization barrier
  - Many others that we will not cover in the class



x

Basic Arm Instructions

- Instructions consist of several fields that encode the opcode and arguments to the opcode
- Special fields enable extended functionality - later
- Several 4-bit operand fields for specifying the source and destination of the operation, usually one of the 16 registers
- Embedded constants *("immediate values")* of various size and "configuration"
- Basic Data processing instruction formats (below)

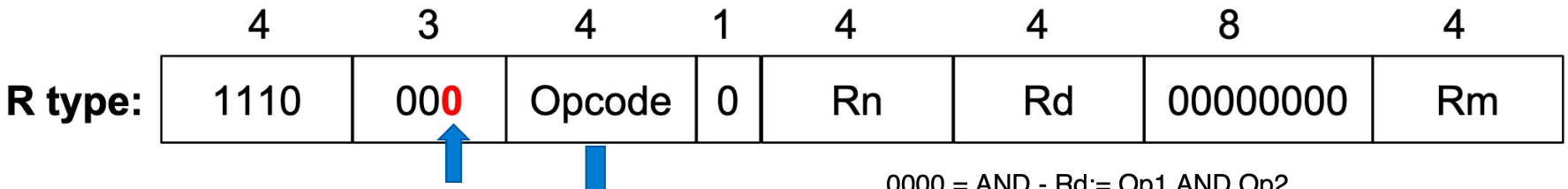


x

R (register) -Type Data Processing

- Instructions that process data using three-register arguments
- The general instruction format is (not all fields will be in every instruction)

```
opcode   Rd (destination), Rn (operand 1), Rm (operand 2)
```




```
add r0, r1, r3
```

is encoded as

1110 0000 1000 0001 0000 0000 0000 0011

in hex is

0xe0810003

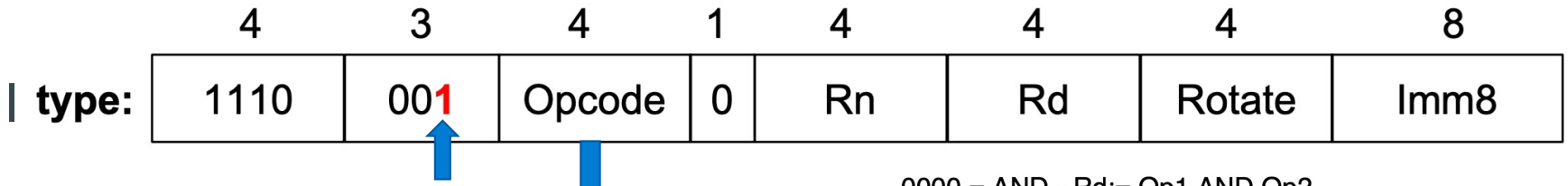
0000 = AND - Rd:= Op1 AND Op2
0001 = EOR - Rd:= Op1 EOR Op2
0010 = SUB - Rd:= Op1 - Op2
0011 = RSB - Rd:= Op2 - Op1
0100 = ADD - Rd:= Op1 + Op2
0101 = ADC - Rd:= Op1 + Op2 + C
0110 = SBC - Rd:= Op1 - Op2 + C - 1
0111 = RSC - Rd:= Op2 - Op1 + C - 1
1000 = TST - set condition codes on Op1 AND Op2
1001 = TEQ - set condition codes on Op1 EOR Op2
1010 = CMP - set condition codes on Op1 - Op2
1011 = CMN - set condition codes on Op1 + Op2
1100 = ORR - Rd:= Op1 OR Op2
1101 = MOV - Rd:= Op2
1110 = BIC - Rd:= Op1 AND NOT Op2
1111 = MVN - Rd:= NOT Op2

x

I (immediate) - Type Data Processing

- Instructions that process data using two registers and a constant (in the instruction)
- The general instruction format is (not all fields will be in every instruction)

```
opcode   Rd (destination), Rn (operand 1), constant
```




```
add r0, r1, 49
```

is encoded as

1110 0000 1000 0001 0000 0000 0011 0001

in hex is

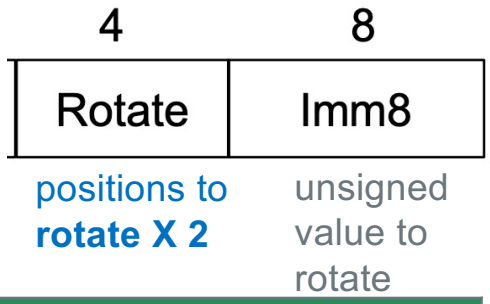
0xe0810031

0000 = AND - Rd:= Op1 AND Op2
0001 = EOR - Rd:= Op1 EOR Op2
0010 = SUB - Rd:= Op1 - Op2
0011 = RSB - Rd:= Op2 - Op1
0100 = ADD - Rd:= Op1 + Op2
0101 = ADC - Rd:= Op1 + Op2 + C
0110 = SBC - Rd:= Op1 - Op2 + C - 1
0111 = RSC - Rd:= Op2 - Op1 + C - 1
1000 = TST - set condition codes on Op1 AND Op2
1001 = TEQ - set condition codes on Op1 EOR Op2
1010 = CMP - set condition codes on Op1 - Op2
1011 = CMN - set condition codes on Op1 + Op2
1100 = ORR - Rd:= Op1 OR Op2
1101 = MOV - Rd:= Op2
1110 = BIC - Rd:= Op1 AND NOT Op2
1111 = MVN - Rd:= NOT Op2

x

How are I – Type Constants Encoded in the instruction?

- Aarch32 provides only 8-bits for specifying an immediate constant value
- This does not provide a very large range of constant values (0-255)
- Imm8 expands to 32 bits and does a rotate right to achieve additional constant values



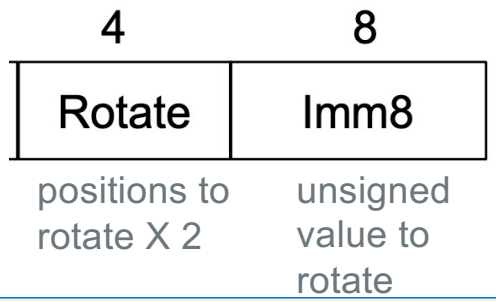
rot4 value	32-bit constant result
0000	0000 0000 0000 0000 0000 0000 1111 1111
0001 (2 bits)	1100 0000 0000 0000 0000 0000 0011 1111

results are interpreted as a 2's complement number

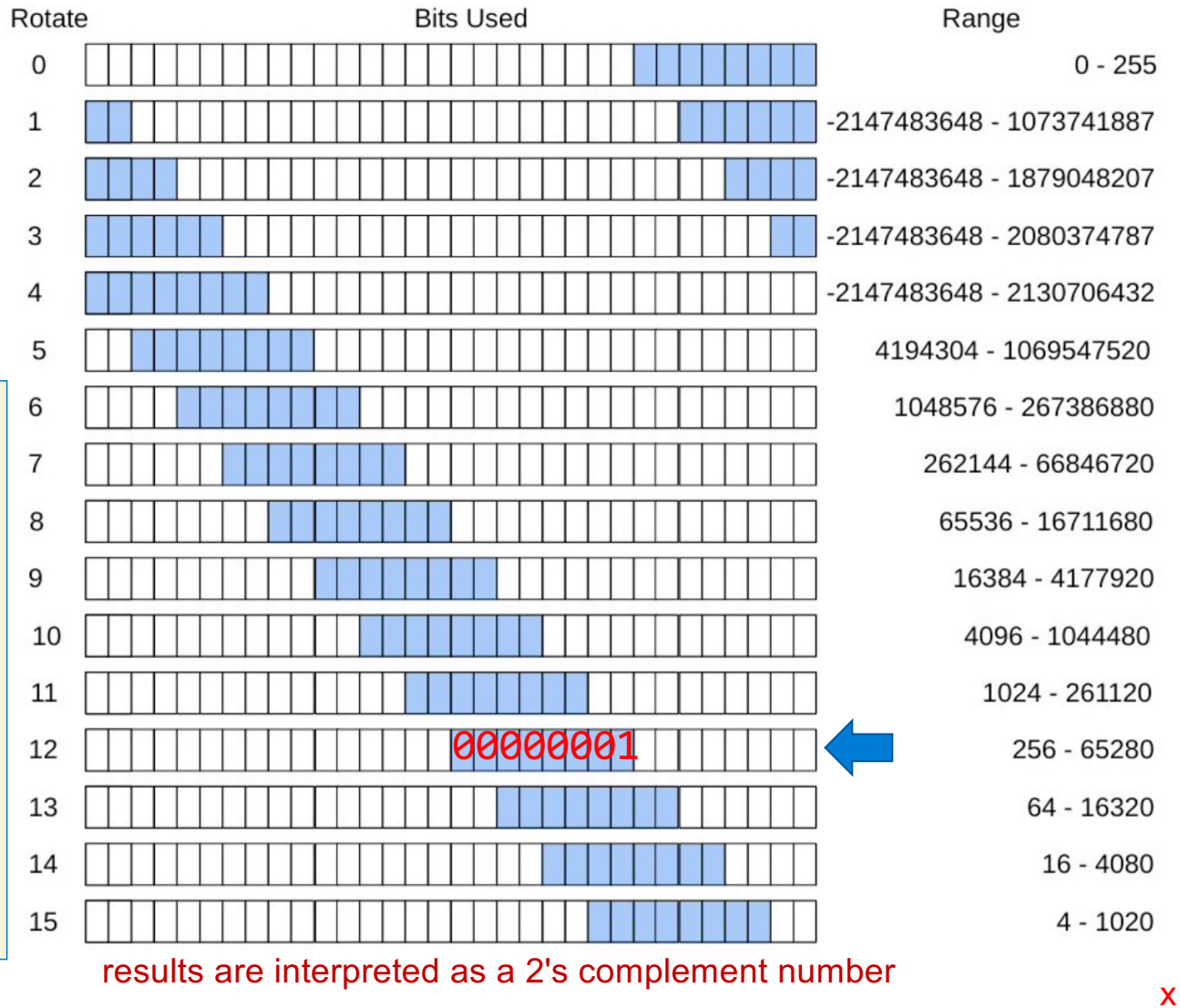
two bits rotated right

x

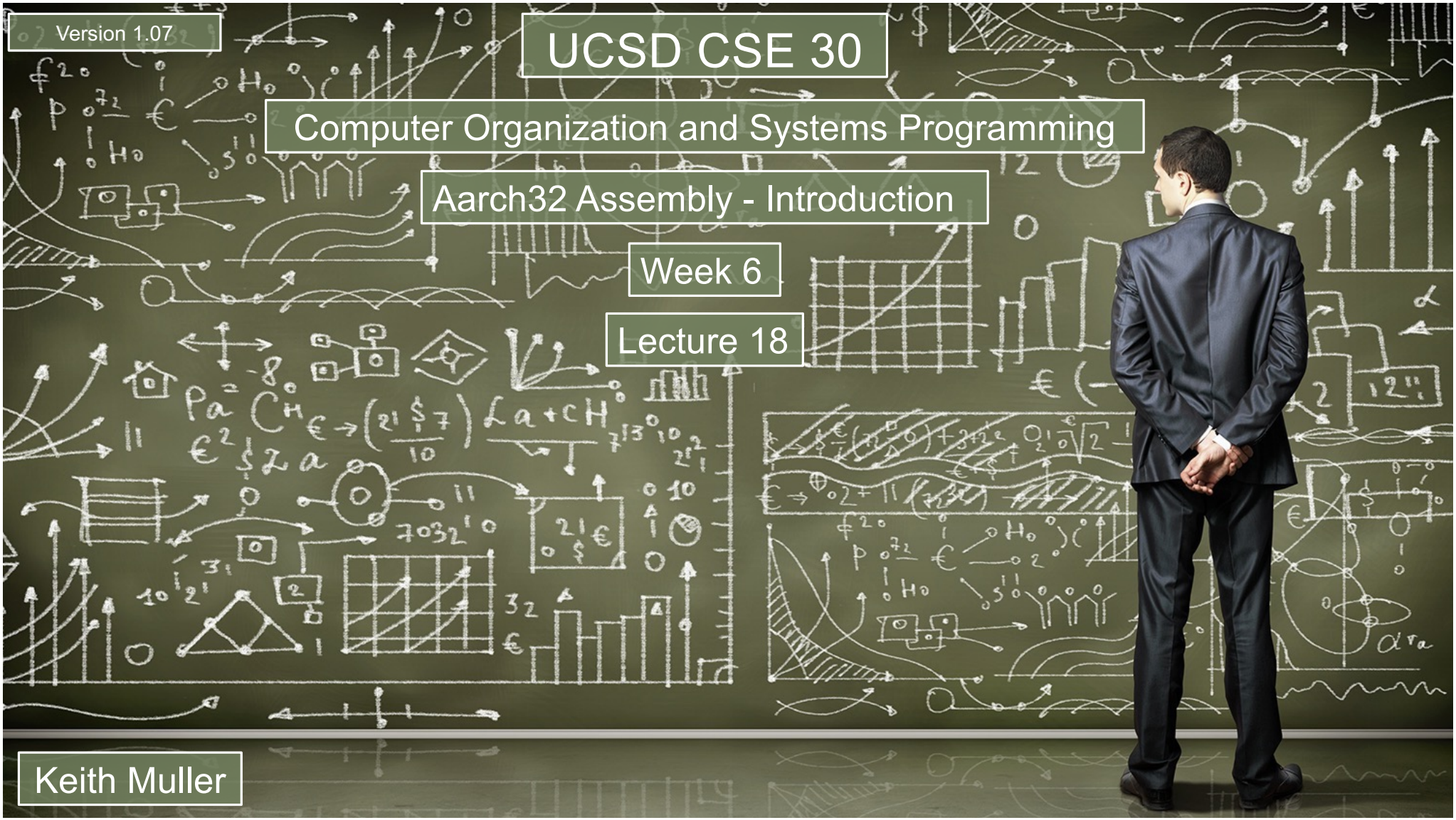
Rot4 - Imm8 Values



- How would 256 be encoded?
  - rotate = 12, imm8 = 1
- **Bottom line:** the assembler will do this for you
- If you try and use an immediate value that it cannot generate it will give an error
- There is a workaround - later

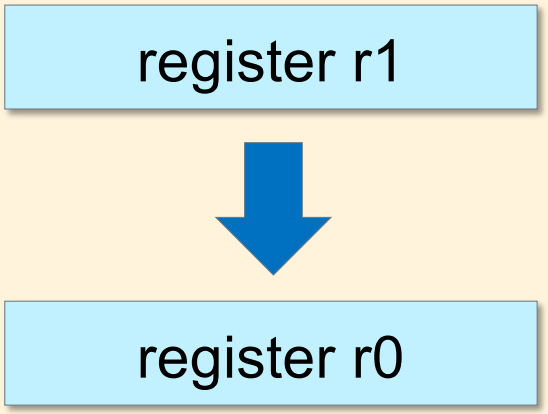


x

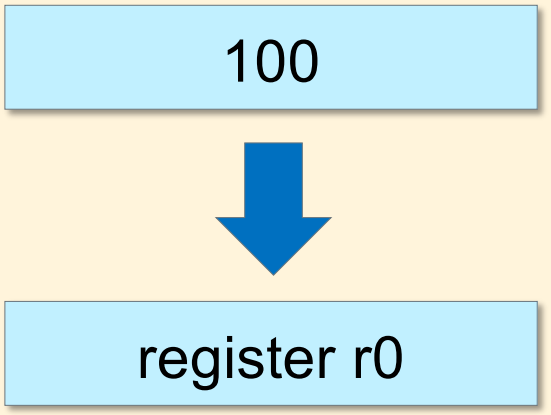

Version 1.07
UCSD CSE 30
Computer Organization and Systems Programming
Aarch32 Assembly - Introduction
Week 6
Lecture 18
Keith Muller

First Look: Copying Values To Registers - MOV

```
mov   r0, r1

// Copies all 32 bits
// of the value held
// in register r1 into
// the register r0
```

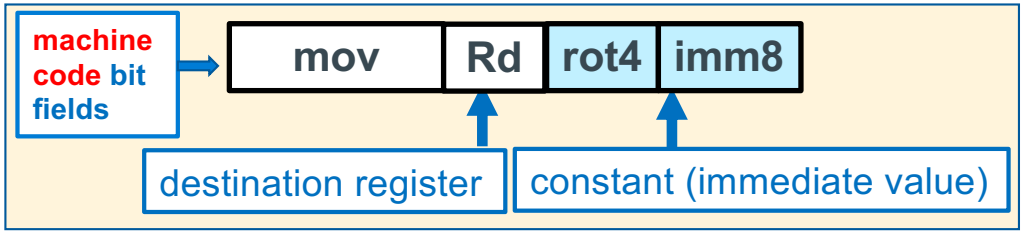


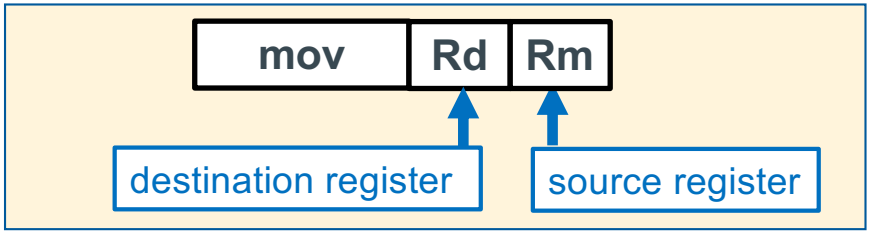

```
mov   r0, 100

// Expands an imm8 value 100
// stored in the instruction
// into the register r0
```

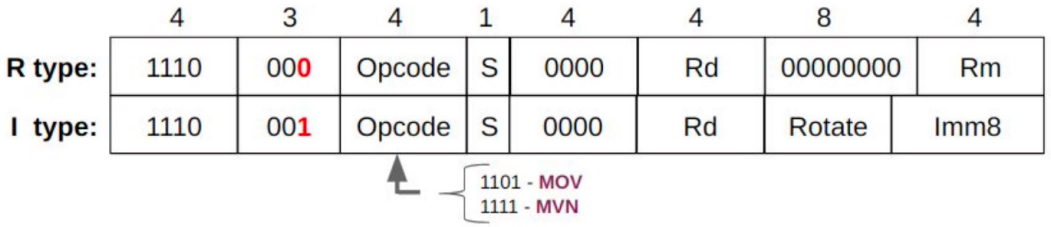


x

mov – Copies Register Content between registers





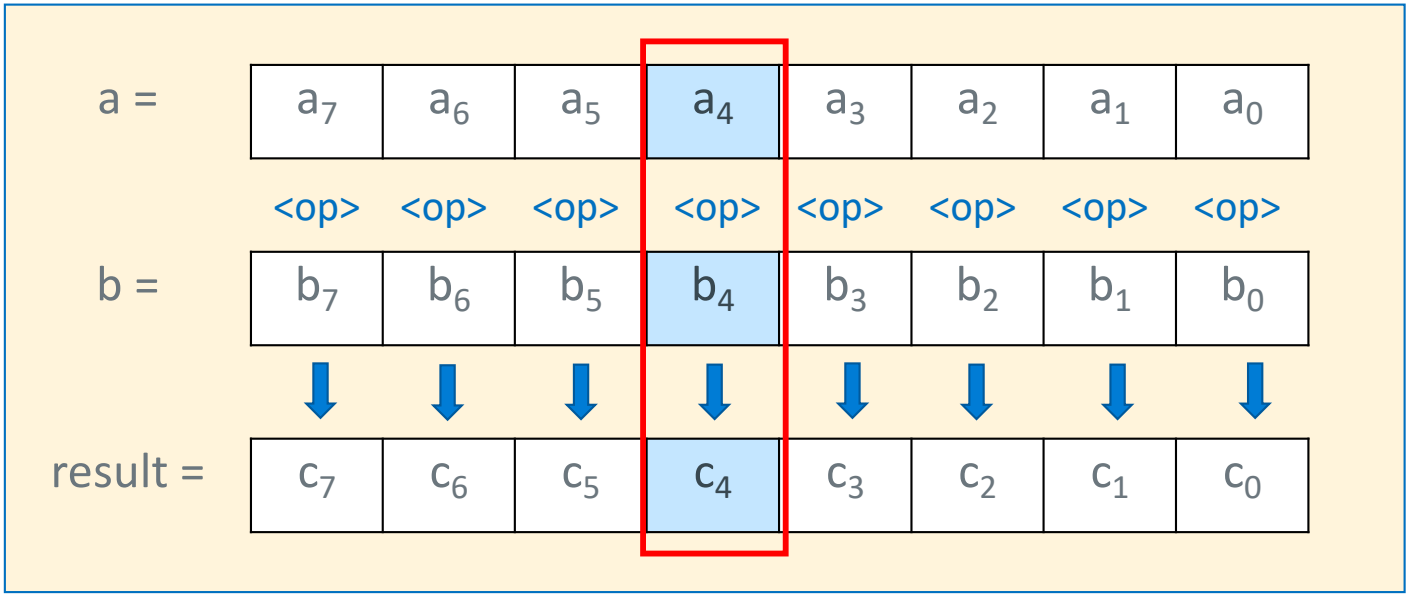



```
mov  Rd, constant   // Rd = constant
mov  Rd,  Rm        // Rd = Rm
```

```
mov  r1, r5         // r1 = r5
mov  r1, 1          // r1 = 1
mov  r1, -4         // r1 = -4
```


x

What is a Bitwise Operation?



- Bitwise operators are applied to each of the corresponding bit positions in each variable
- Each bit position of the result depends only on bits in the same bit position within the operands

x

Bitwise Not (vs Boolean Not)

```
in C
int output = ~a;
```

a	~a
0	1
1	0

Bitwise NOT

```
~ 1100
  ----
  0011
```

	Bitwise Not
number	0101 1010 0101 1010 1111 0000 1001 0110
~number	1010 0101 1010 0101 0000 1111 0110 1001

Meaning	*Operator*	*Operator*	*Meaning*
Boolean NOT	!b	~b	Bitwise NOT

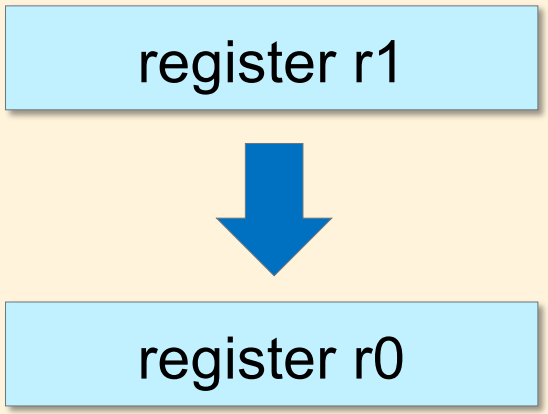
Boolean operators act on the entire value not the individual bits

Type	*Operation*	result
bitwise	~0x01	1111 1111 1111 1111 1111 1111 1111 1110
Boolean	!0x01	0000 0000 0000 0000 0000 0000 0000 0000

x

First Look: Copying Values To Registers – MVN (negate)

```
mvn   r0, r1

// Copies all 32 bits
// of the value held
// in register r1 into
// the register r0
// then does a bitwise NOT
```

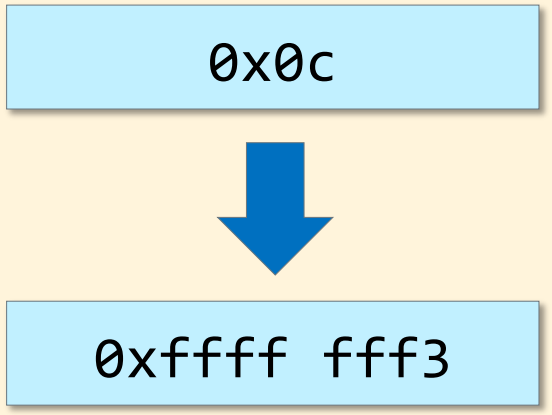


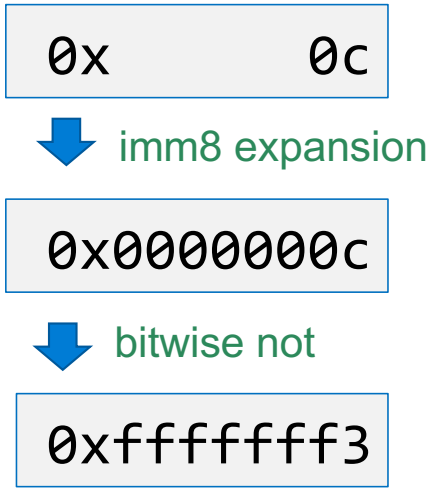
Bitwise NOT

```
~ 1100
  ----
  0011
```

- A **bitwise NOT** operation

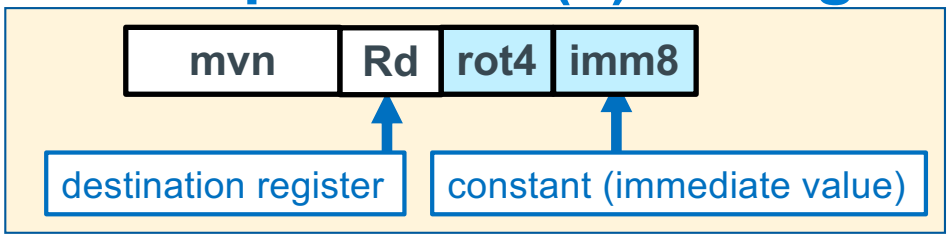
```
mvn   r0, 12

// Expands an imm8 value 0x0c
// stored in the instruction
// into a register then does
// a bitwise NOT
```

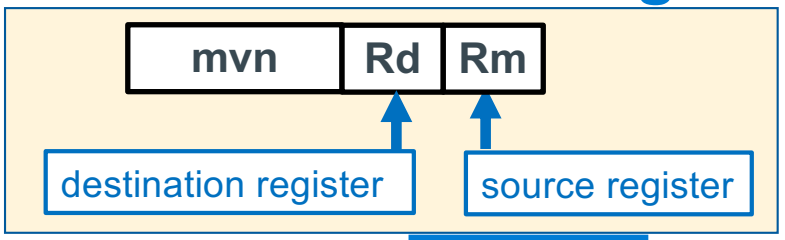




x

mvn – Copies NOT (~) of Register content between registers






```
mvn   Rd, constant    // Rd = constant
mvn   Rd,  Rm         // Rd = Rm
```

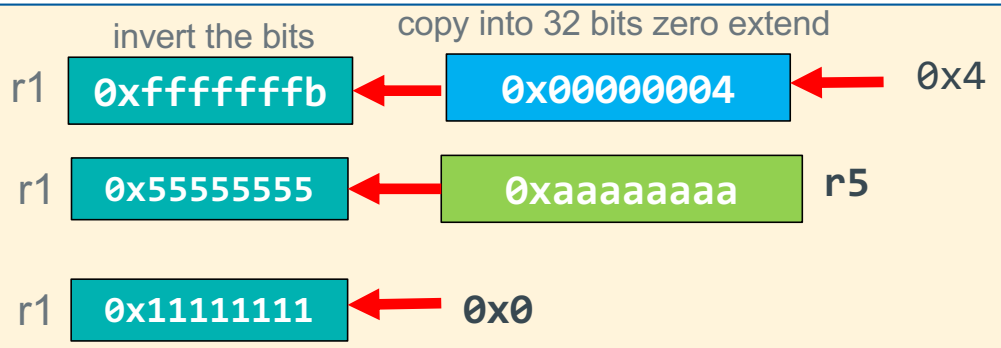
```
~ 1100
  ----
  0011
```

Bitwise NOT

bitwise NOT operation. Immediate (constant) version copies to 32-bit register, then does a bitwise NOT

imm8	extended imm8	inverted imm8	signed base 10
0x00	0x00 00 00 00	0xff ff ff ff	-1
0xff	0x00 00 00 ff	0xff ff ff 00	-256

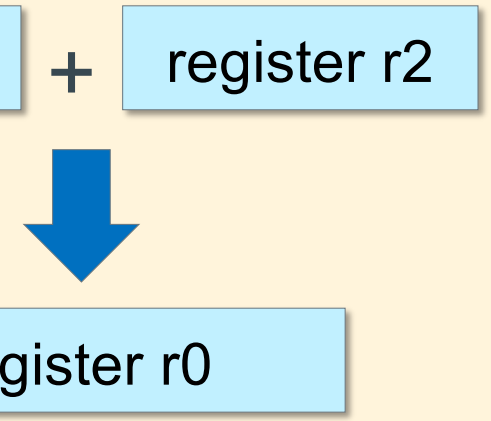
```
mvn    r1, 4          // x = ~4
mvn    r1, r5         // x = ~y in C
mvn    r1, 0          // x = -1
```



x

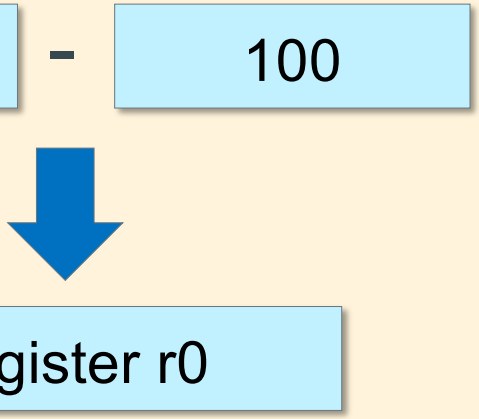
First Look: Add/Sub Registers

```
add   r0, r1, r2
```




```
// Adds r1 to r2 and
// stores the result
// in r0
```

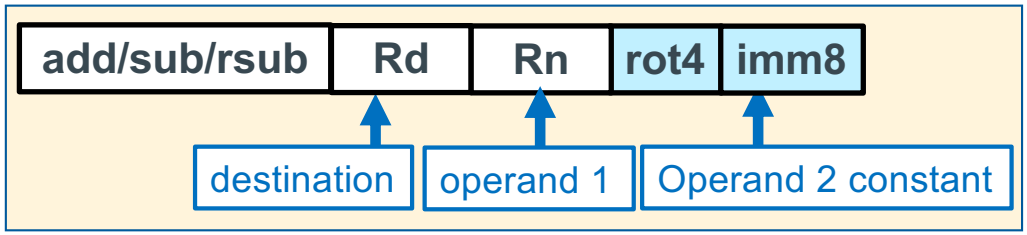
```
sub   r0, r1, 100
```

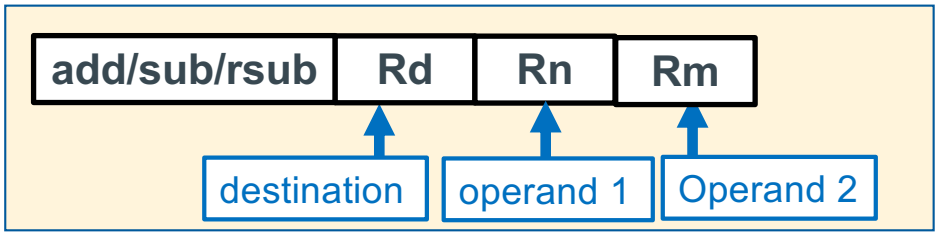



```
// Perform r1 – 100 and
// stores the result in
// r0
```


x

add/sub/rsub – Add or Subtract two integers






```
add  Rd, Rn, constant    // Rd = Rn + constant
sub  Rd, Rn, constant    // Rd = Rn - constant
rsub Rd, Rn, constant    // Rd = constant - Rn
add  Rd,  Rn,  Rm        // Rd = Rn + Rm
sub  Rd,  Rn,  Rm        // Rd = Rn - Rm
rsub Rd,  Rn,  Rm        // Rd = Rm - Rn
```

```
mov   r5, 5        // r5 = 5
mov   r7, 7        // r7 = 7
add   r7, r7, r5   // r7 = 12 r5 = 5
```

```
add   r1, r2, r3   // r1 = r2 + r3
sub   r1, r1, 1    // r1 = r1 – 1; or r1--
add   r1, r2, 234  // r1 = r2 + 234
```


x

Writing a Sequence of Add & Subtract Instructions

- You need to perform the following sequence of integer adds/subtracts

```
a = b + c + d - e;
```

- Since ARM uses a three-operand instruction set, you can only operate on two operands at a time
- So, you need to use one register as an **accumulator** and create **a sequence of add instructions** to build up the solution

```
r0 ← a
r1 ← b
r2 ← c
r3 ← d
r4 ← e
```

```
a  = b  + c  + d  - e;
r0 = r1 + r2 + r3 - r4;
r0 = ((r1 + r2) + r3) - r4;
r0 = r1 + r2;
r0 = r0 + r3
r0 = r0 - r4
```

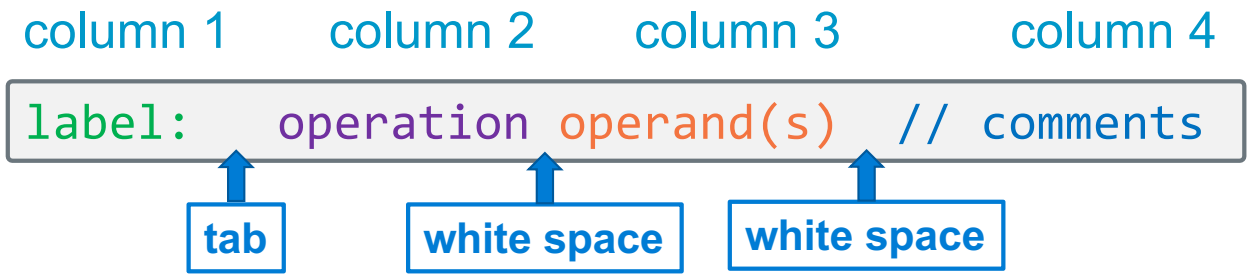
```
add   r0, r1, r2
add   r0, r0, r3
sub   r0, r0, r4
```

```
a = (b + c) - 5;
r0 = (r1 + r2) - 5;
```

```
add   r0, r1, r2
sub   r0, r0, 5
```


x

Overview: Line Layout in an Arm Assembly Source File - 1



- Assembly language source text files are **line oriented** (each ending in a '\n')
- **Each line represents** a **starting address in memory** and does **one** of:
  1. Specifies the contents of memory for a variable (segments containing data)
  2. Specifies the contents of memory for an instruction (text segment)
  3. **Assembler directives** tell the assembler to do something (for example, change label scope, define a macro, etc.) that does not allocate memory
- **Each line** is **organized into up to four columns**
  - Not every column is used on each line
  - Not every line will result in memory being allocated

x

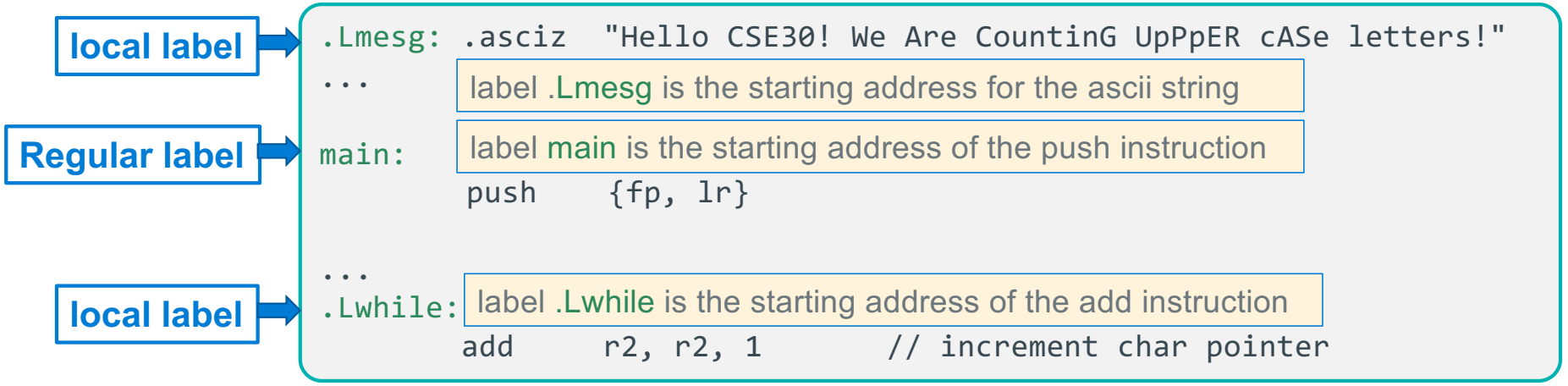
Overview: Line Layout in an Arm Assembly Source File - 2

```
label:  operation operand(s)  // comment

        // assembler directive below
cnt:    .word 5                  /* define a global int cnt = 5;

        /* instruction below */
        add    r1   r2, r3       // add the values
```

1. **Labels** (optional); starts in column 1
   - **Only used** when you need to associate a name to a starting location in memory; You can then refer to the address **by name** in an instruction
2. **Operation type 1**: **assembler directives** (all start with a period e.g. `.word`)
3. **Operation Type 2**: assembly language **instructions**
4. **Zero or more operands** as required by the instruction or assembler directive
5. Comments C and C++ style; also @ in the place of a C++ comment //

x

Labels in Arm Assembly



- Remember, a Label associates a name with memory location
- **Regular Label:**
  - Used with a Function name (label) or for static variables in any of the data segments
- **Local Label:** Name starts with `.L` (local label prefix) only usable in the same file
  1. Targets for branches (if), switch, goto, break, continue, loops (for, while, do-while)
  2. Anonymous variables (string not foo in `char *foo = "anonymous variable"`)
  3. Read only literals when allocated in the text segment – special case)

x

Assembler Directives: Label Scope Control (Normal Labels only)

```
.extern printf
.extern fgets
.extern strcpy
.global fbuf
```

`.extern <label>`

- **Imports** label (function name, symbol or a static variable name);
- An address associated with the label from another file can be used by code in this file

`.global <label>`

- **Exports** label (or symbol) to be visible outside the source file boundary (other assembly or c source)
- label is either a function name or a global variable name
- **Without** .global, **by default** labels are local to the file from the point where they are defined

x

Assembler Directives: .equ and .equiv

```
        .equ    BLKSZ,  10240       // buffer size in bytes
        .equ    BUFCNT, 100*4       // buffer for 100 ints
        .equiv  STRSZ,  128         // buffer for 128 bytes
        .equiv  STRSZ,  1280        // ERROR! already defined!
        .equ    BLKSZ,  STRSZ * 4 // redefine BLKSZ from here
```

`.equ <symbol>, <expression>`

- Defines and sets the value of a symbol to the evaluation of the expression
- Used for specifying constants, like a `#define` in C
- You can (re)set a symbol many times in the file, last one seen applies

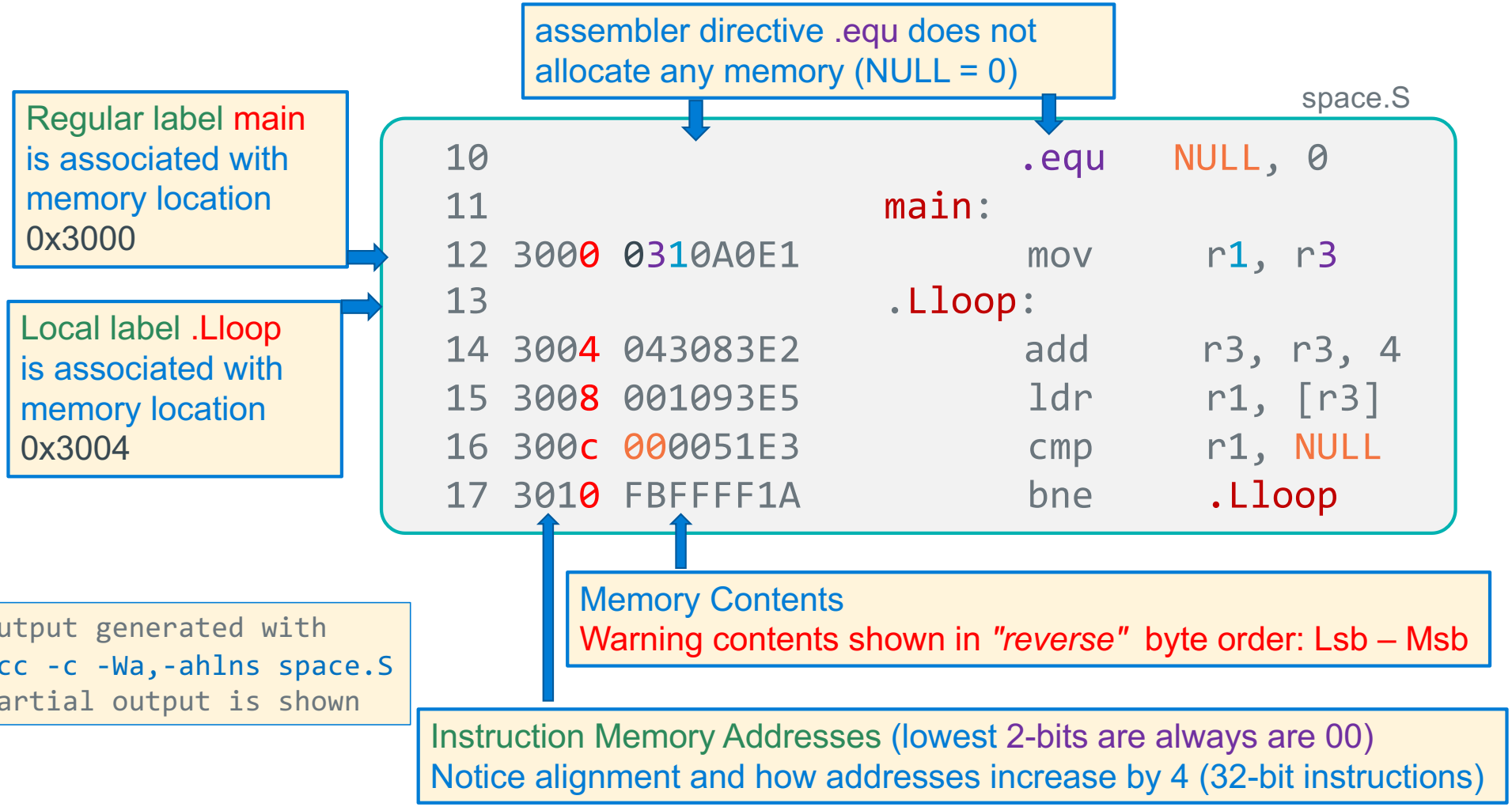
```
        .equ    BLKSZ,  10240       // buffer size in bytes
        // other lines
        .equ    BLKSZ,  1024        // buffer size in bytes
```

`.equiv <symbol>, <expression>`

- `.equiv` directive is like `.equ` except that the assembler will signal an error if symbol is already defined

x

Example: Assembler Directive and Instructions



x