

Version 1.07

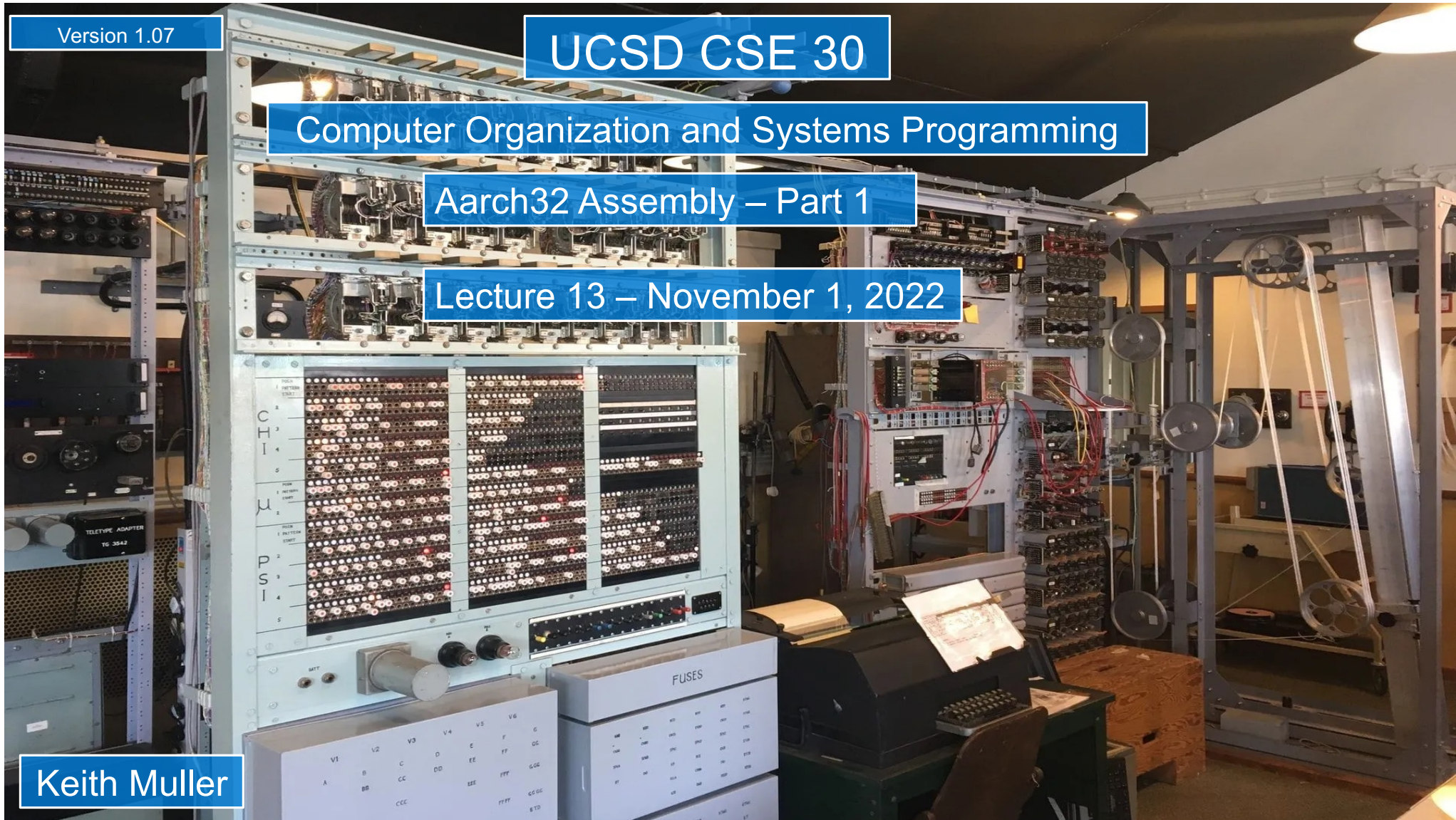
UCSD CSE 30

Computer Organization and Systems Programming

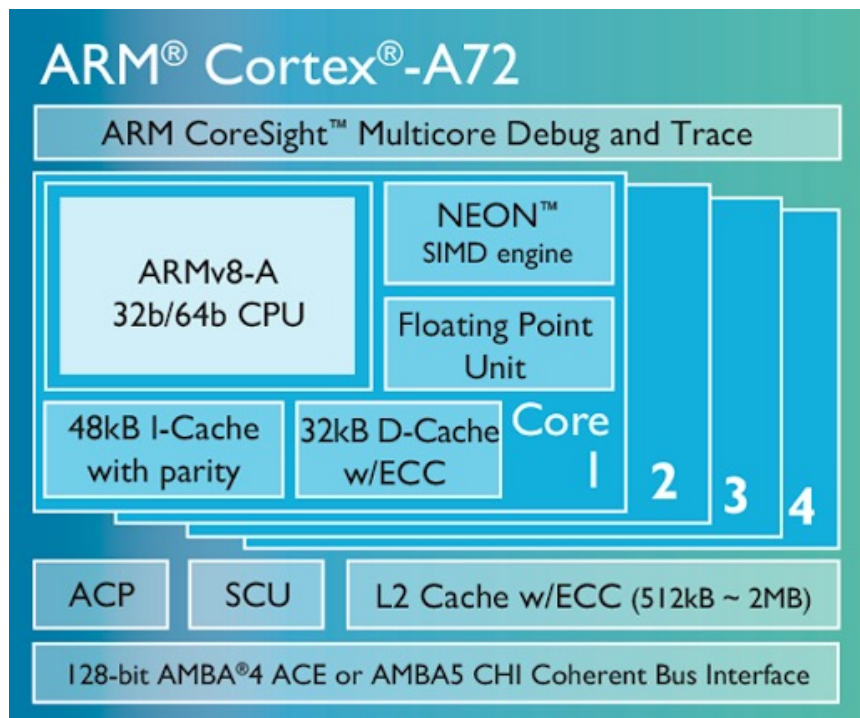
Aarch32 Assembly – Part 1

Lecture 13 – November 1, 2022

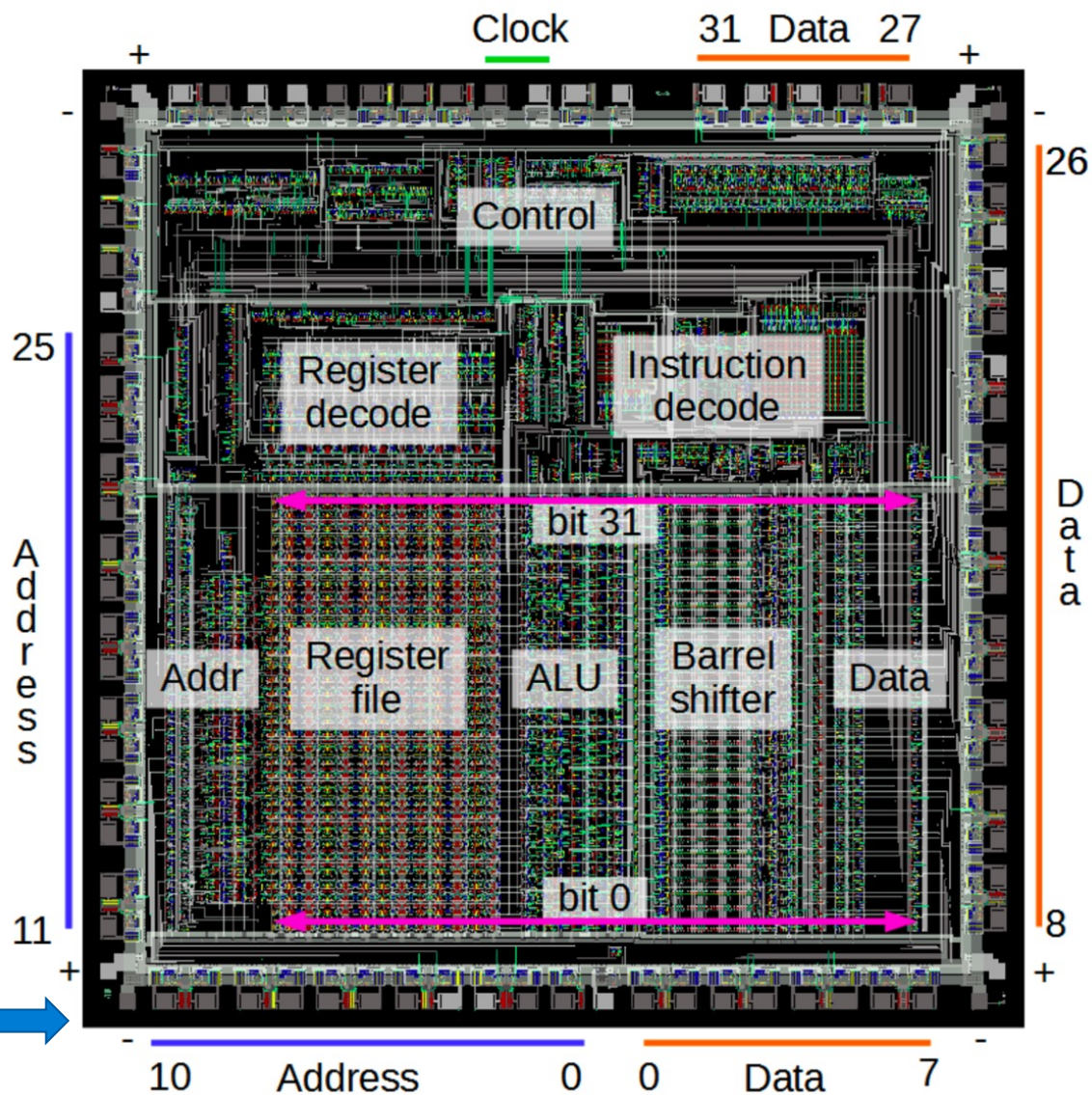
Keith Muller



Arm Core Organization & Floorplan Examples



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



Memory Triangle: Hardware Cost/Performance/Capacity Tiers

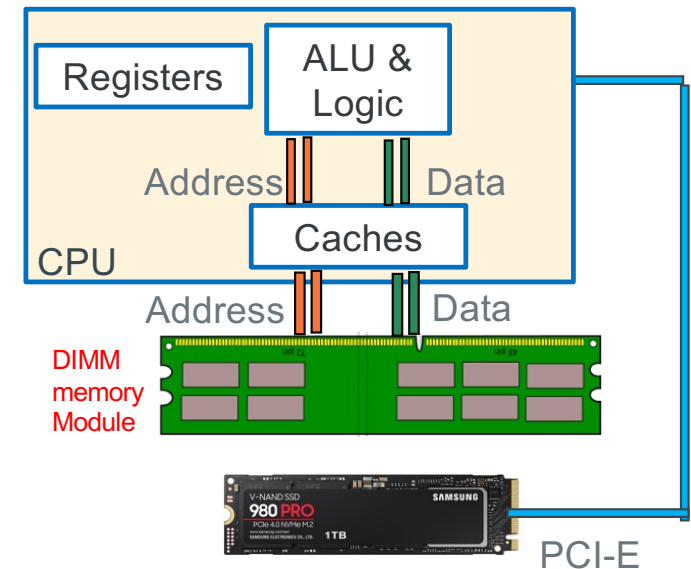
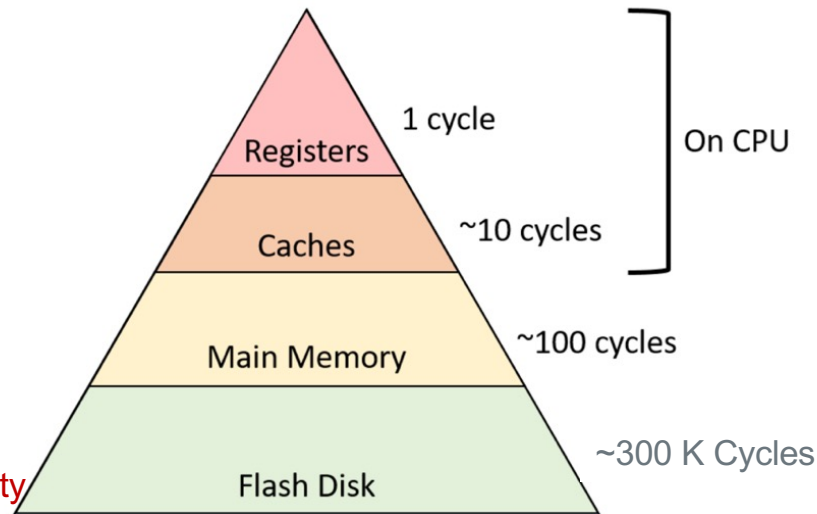
Goal: keep most Data
Accesses high in the
Triangle

1. Smallest Capacity
2. Highest Performance
3. Highest cost/capacity



1. Largest Capacity
2. Slowest performance
3. Lowest Cost \$/capacity

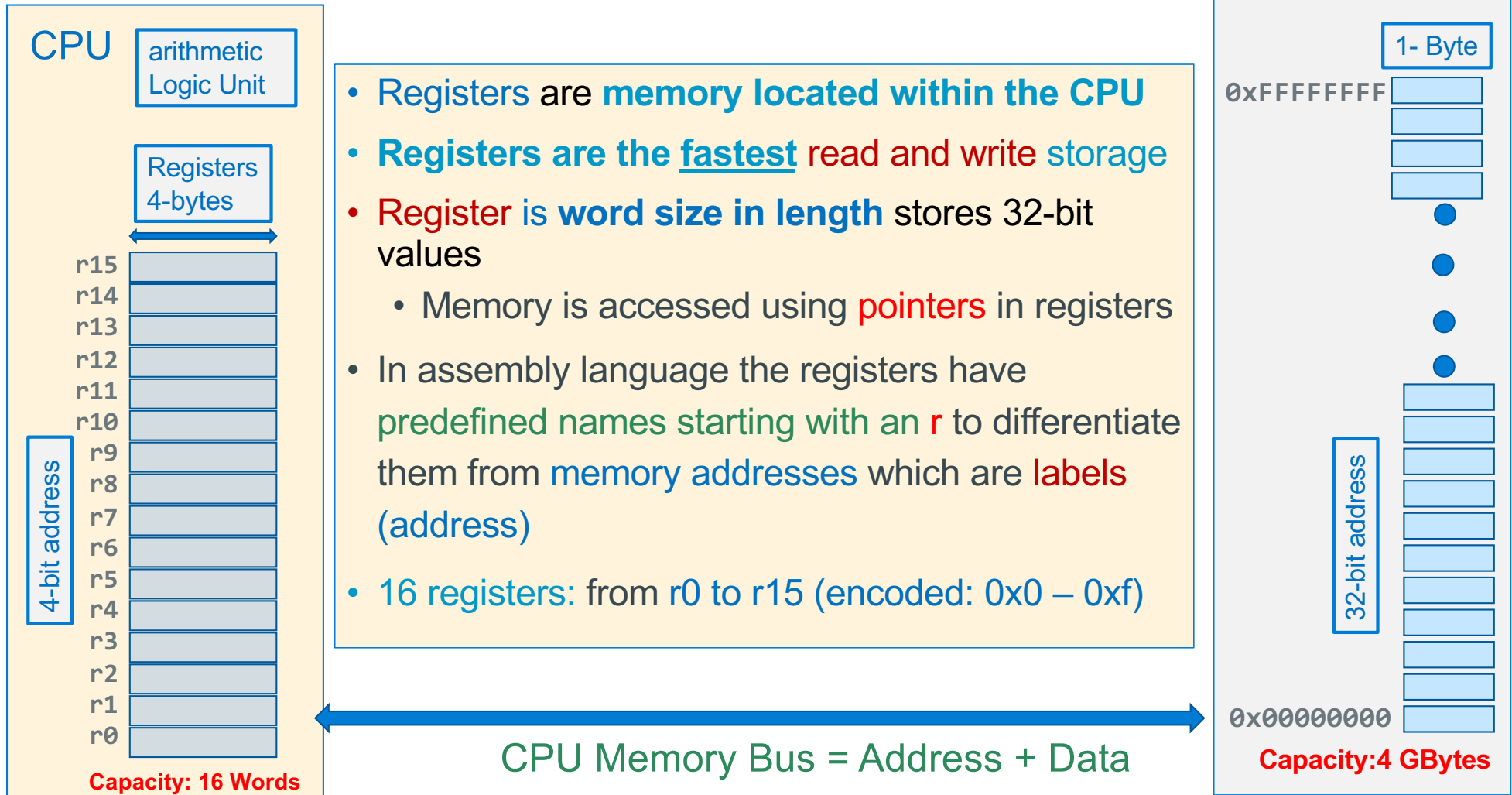
Assume 1 clock cycle
per machine instruction



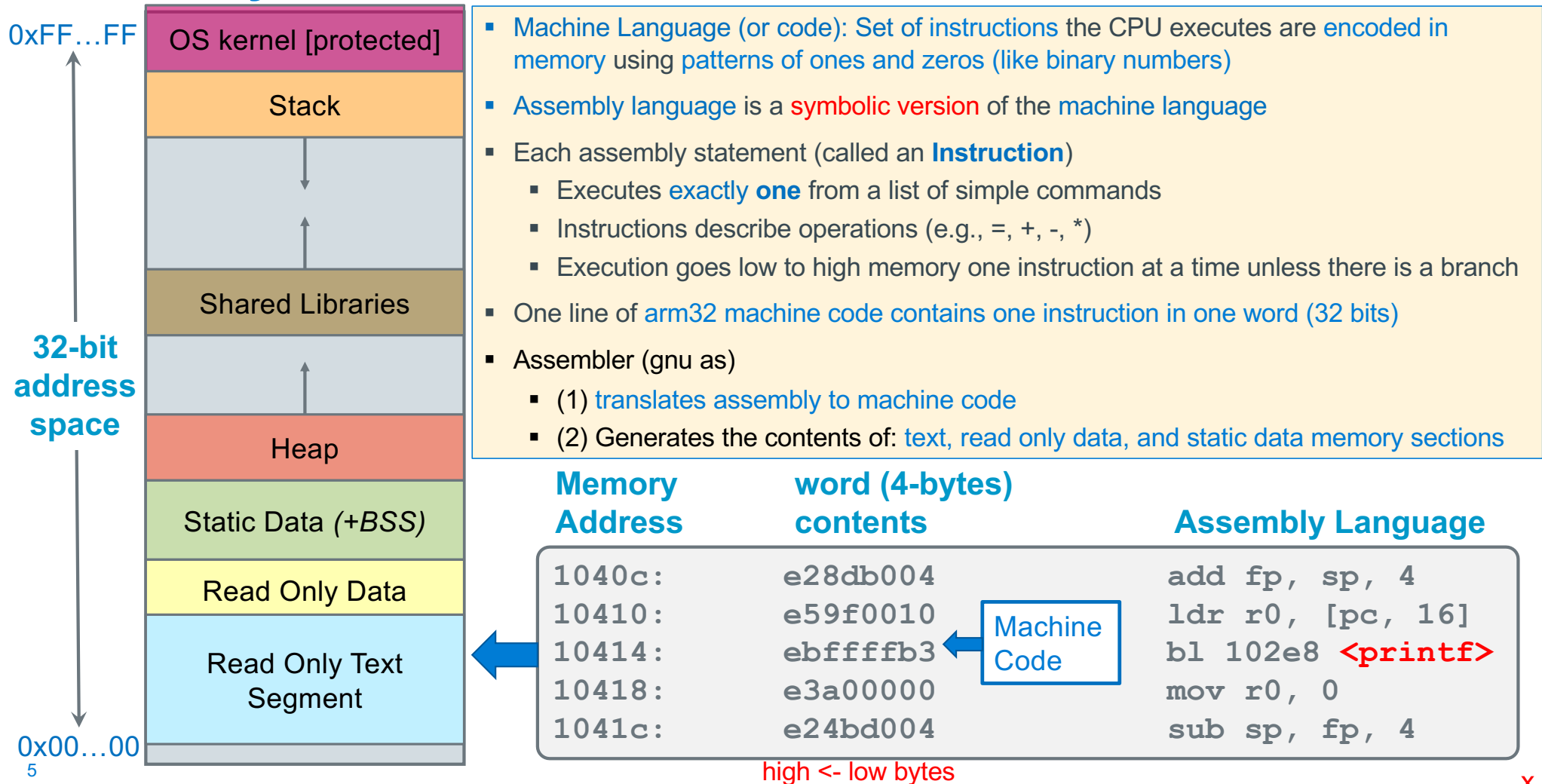
Clock cycle \approx time to access; larger is slower

Design Tradeoff: Based on workload considering cost and performance targets

32-Bit Arm - Registers

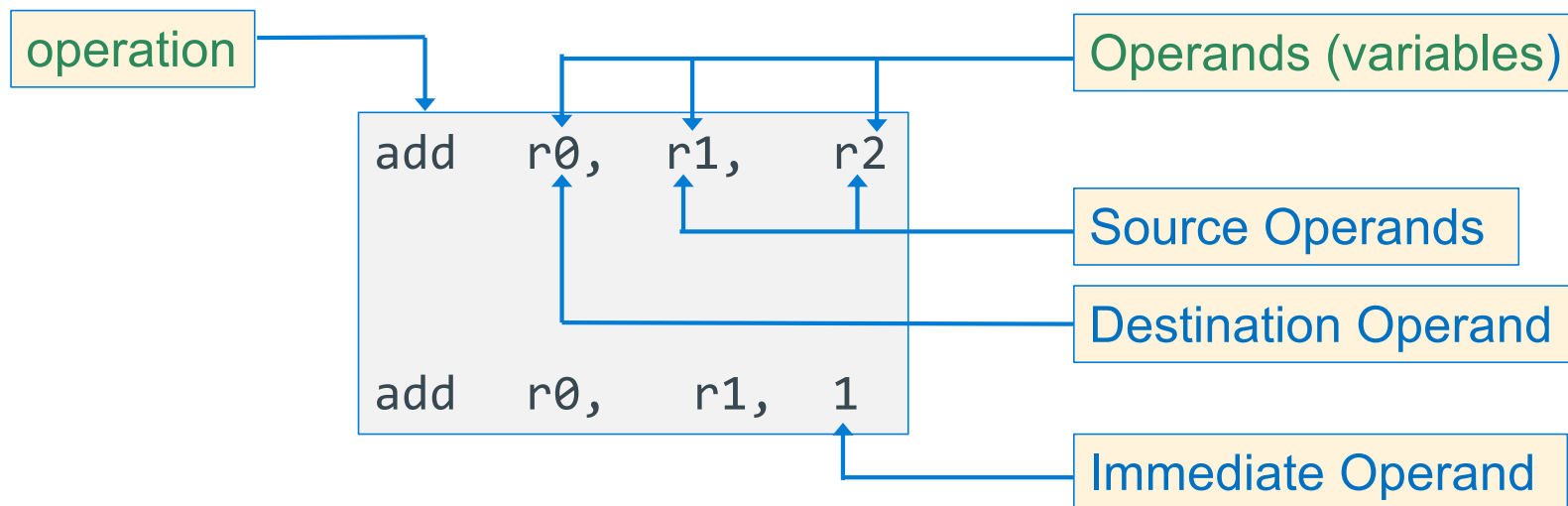


Assembly and Machine Code



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 $y = x + 1$



Meaning of an Instruction

- Operations are abbreviated into **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;           // c
```

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


32-Bit Arm - Registers

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

Arithmetic & Logic Unit (ALU)

registers

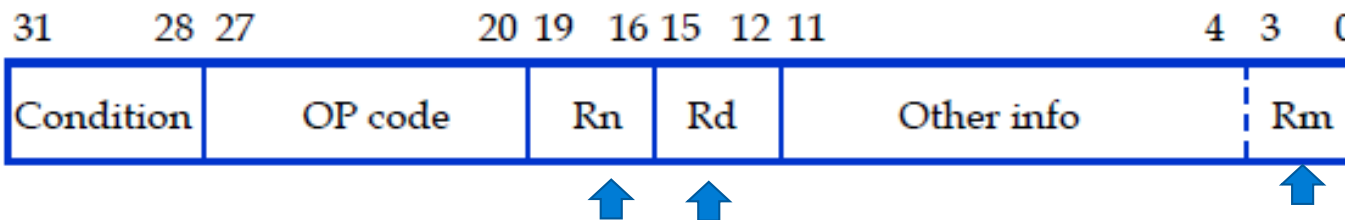
4-bytes

r15
r14
r13
r12
r11
r10
r9
r8
r7
r6
r5
r4
r3
r2
r1
r0

4-bit address

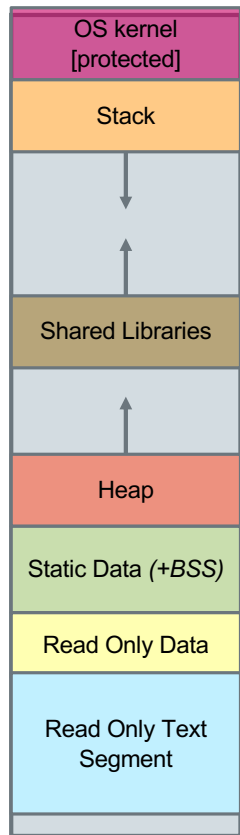
Capacity: 16 Words

- 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)



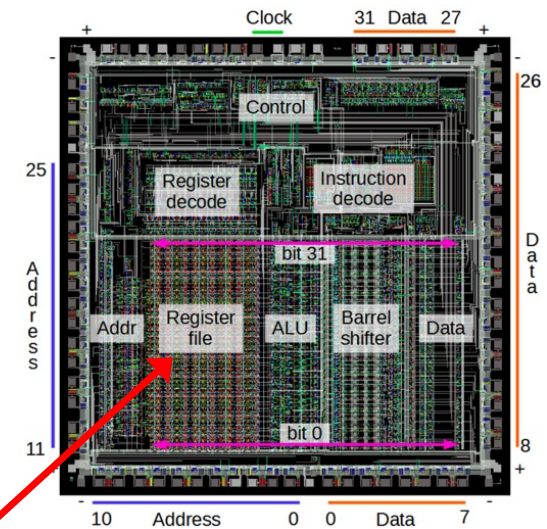
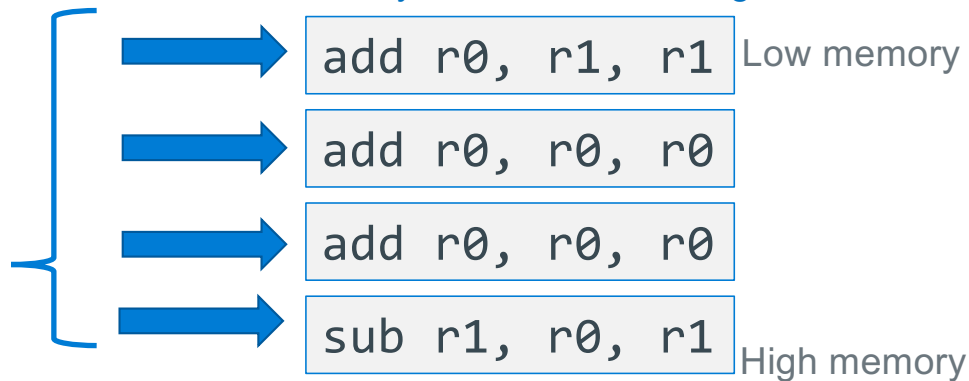
Program Execution: A Series of Instructions

Main Memory



- Instructions are **retrieved sequentially** from memory
- Each instruction **executes to completion before the next instruction is completed**
- Conceptually the pc (program counter) points at executing instruction
- exceptions: loops, function calls, traps,...

Memory Content in Text segment



Register contents inside the CPU

r0 = 1 r1 = 2 initial values

r0 = 4 r1 = 2

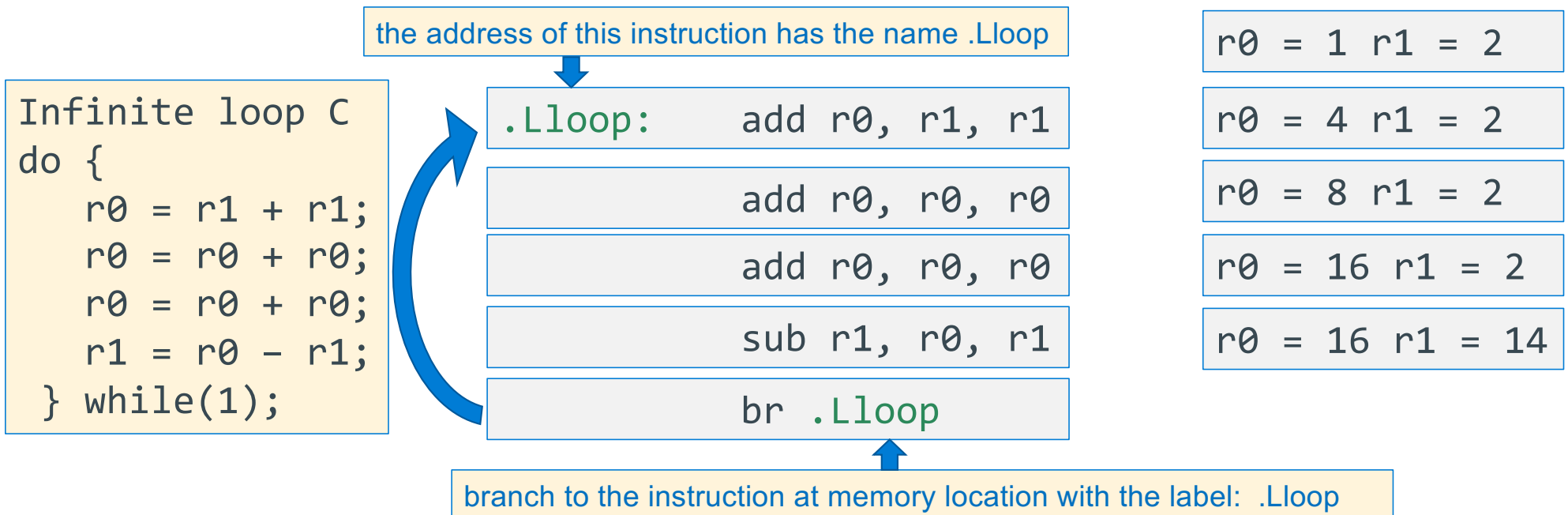
r0 = 8 r1 = 2

r0 = 16 r1 = 2

r0 = 16 r1 = 14

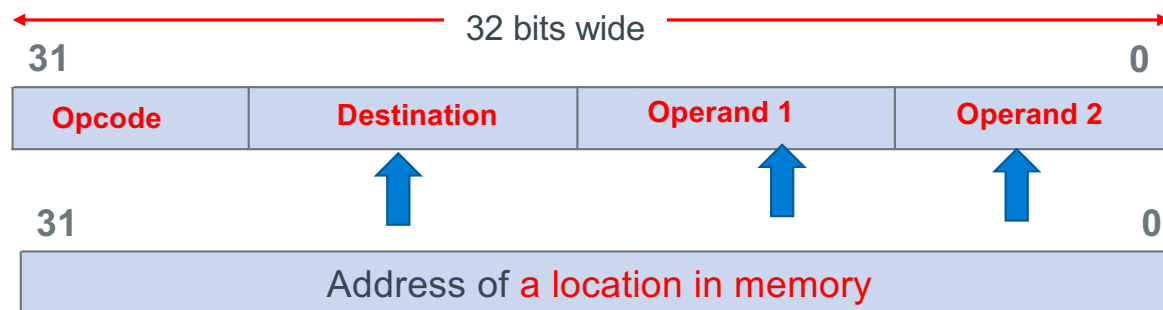
Program Execution: Looping in the Execution Flow

- 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 **infinite** loop (br instruction: unconditional branch: "goto")



How to Access Memory?

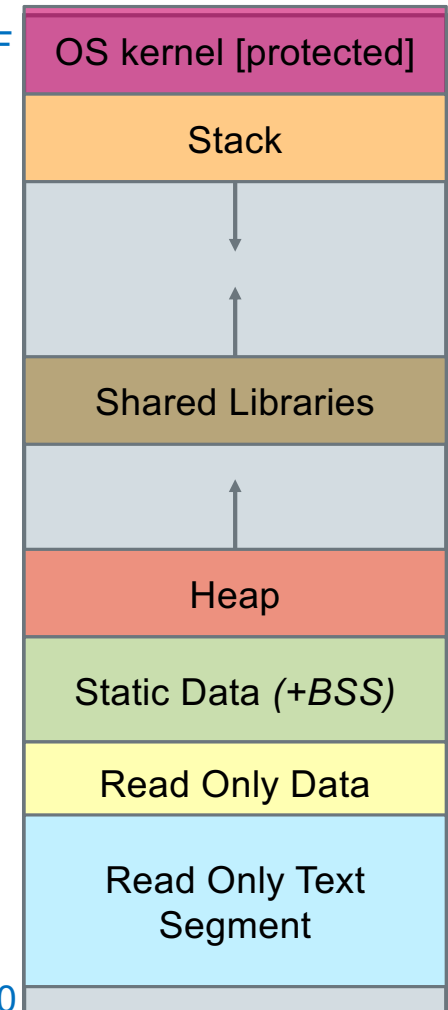
- Consider $a = b + c$ are operands are in memory
 - Operation code: add Destination: a
 - Operand 1: b Operand 2: c
- Aarch32 Instructions are always word size: 32 bits wide
 - Some bits must be used to specify the operation code
 - Some bits must be used to specify the destination
 - Some bits must be used to specify the operands
- Address space is 32 bits wide so put a **POINTER** in a register



0xFF...FF

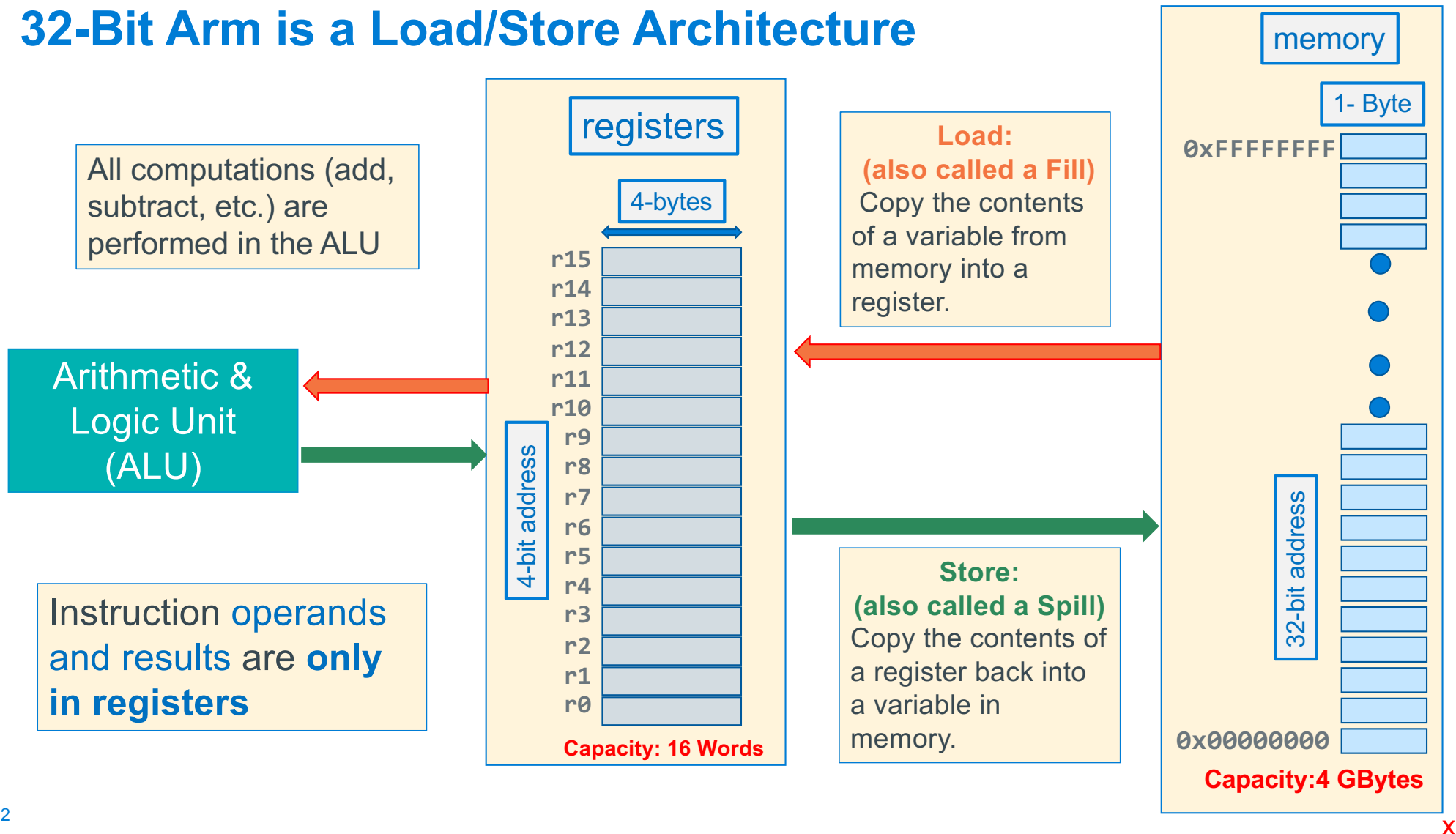
32-bit
Address
space

0x00...00



NOT ENOUGH BITS for FULL Addresses to be stored in the instruction

32-Bit Arm is a Load/Store Architecture



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
```

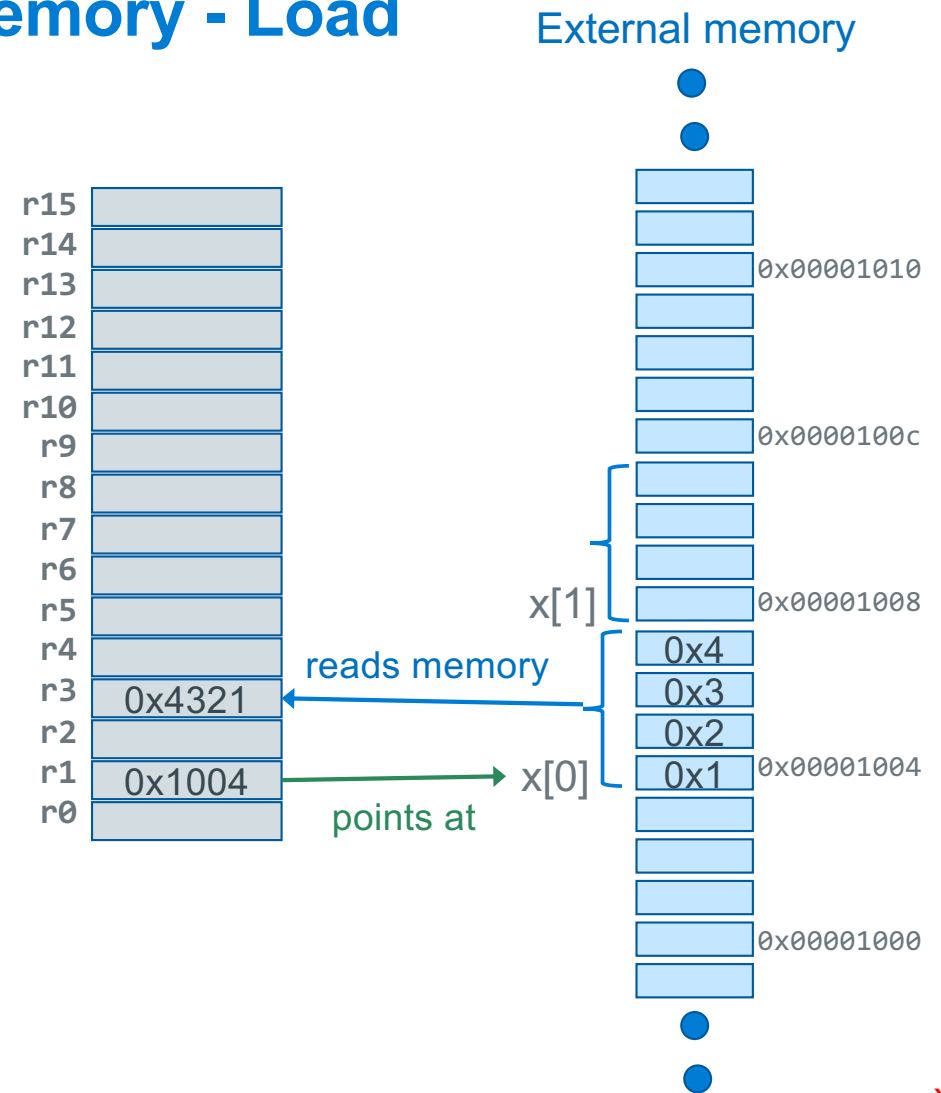
Load register from memory (read)

```
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



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
```

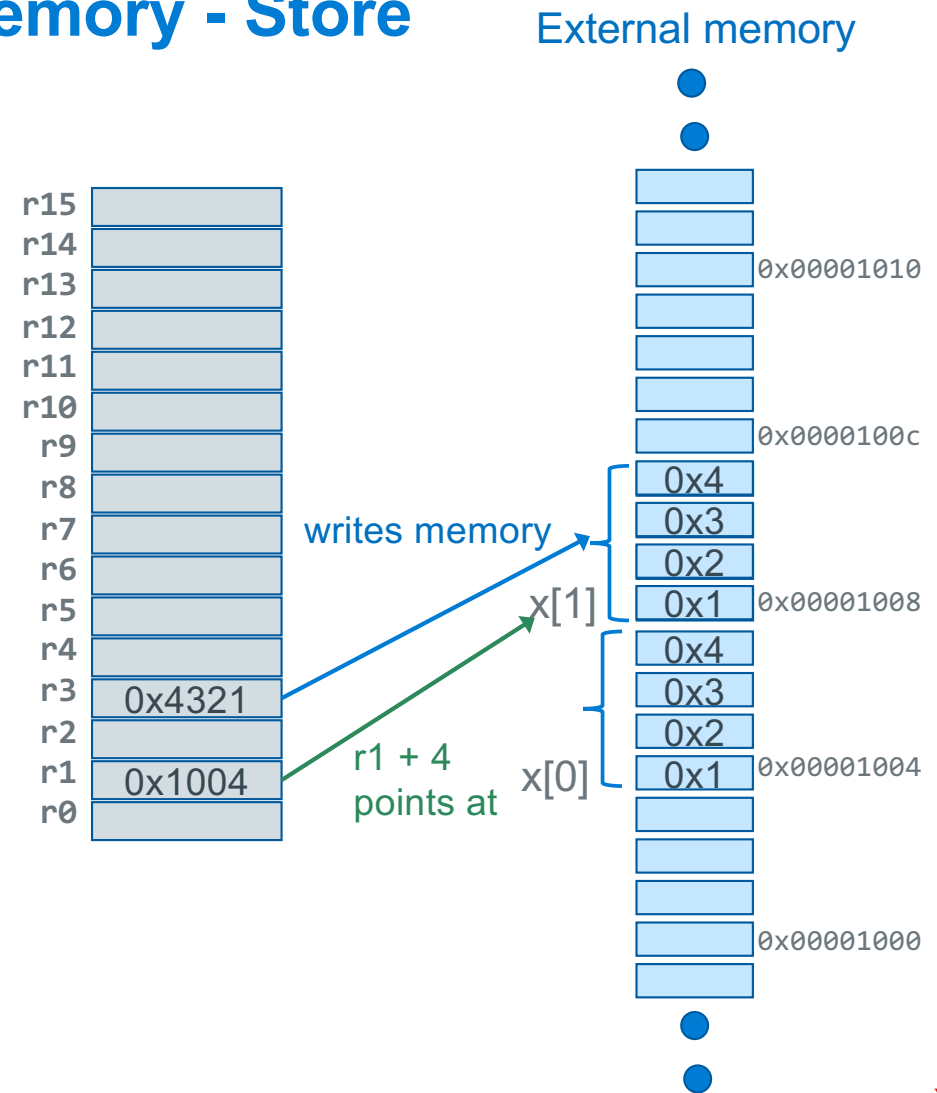
Store register **to** memory (write)

```
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



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

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)

Special Use Registers
program counter

r15/pc

Special Use Registers
function call implementation
& long branching

r14/lr

r13/sp

r12/ip

r11/fp

Preserved registers
Called functions **can't change**

r10

r9

r8

r7

r6

r5

r4

Scratch Registers
First 4 Function Parameters
Function return value
Called functions **can change**

r3

r2

r1

r0

Version 1.07

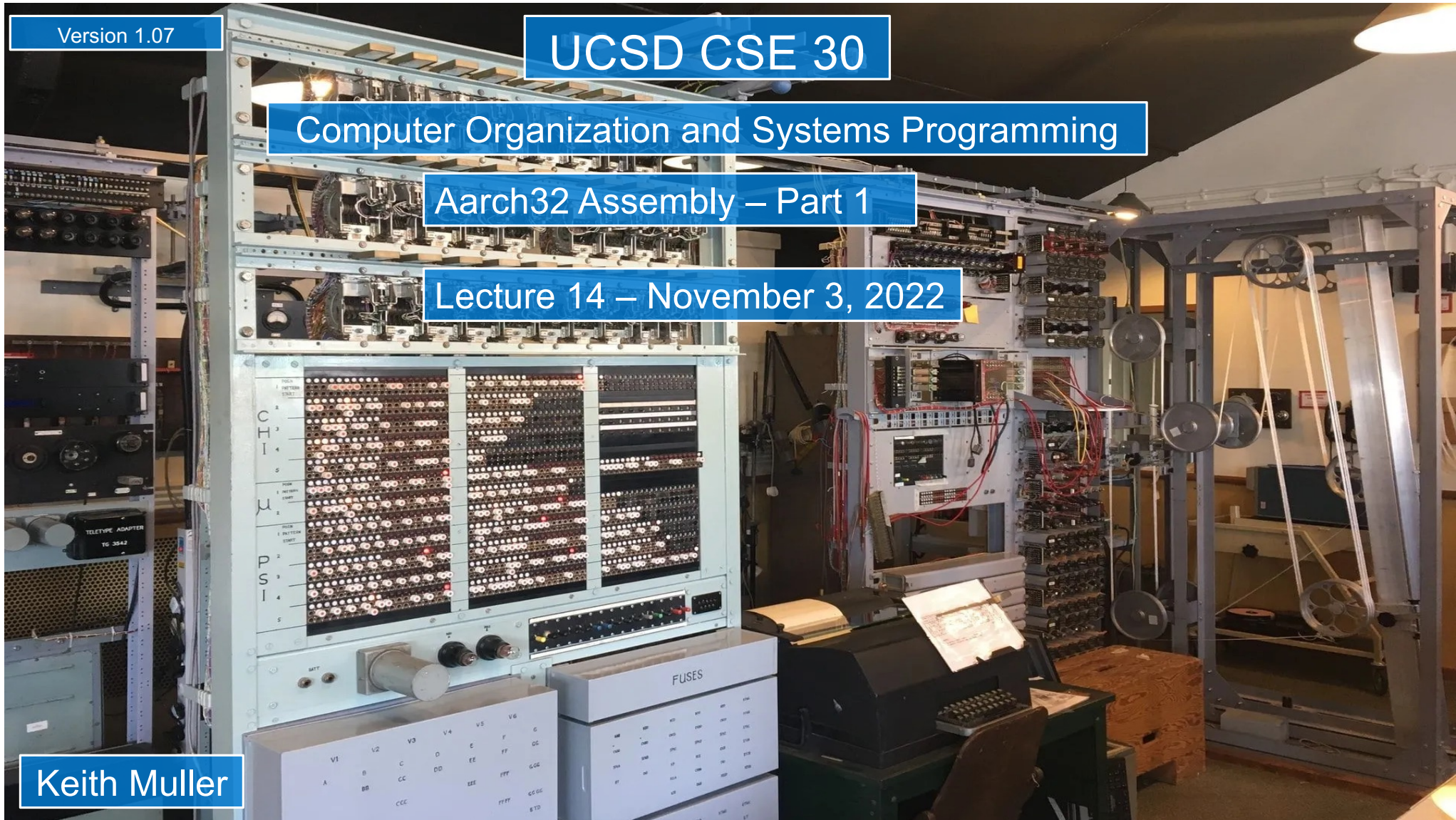
UCSD CSE 30

Computer Organization and Systems Programming

Aarch32 Assembly – Part 1

Lecture 14 – November 3, 2022

Keith Muller



CPU Operational Overview: Executing Machine Code

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**

r15/pc
r15/pc
r15/pc
r15/pc

r15/pc
r15/pc

text segment in memory

address	contents	assembly version
0001042c	<inloop>:	
1042c	e3530061	cmp r3, 0x61
10430	ba000002	blt 10440 <store>
10434	e353007a	cmp r3, 0x7a
10438	ca000000	bgt 10440 <store>
1043c	e2433020	sub r3, r3, #32
00010440	<store>:	
10440	e7c13002	strb r3, [r1, r2]
10444	e2822001	add r2, r2, 0x1
10448	e7d03002	ldrb r3, [r0, r2]
1044c	e3530000	cmp r3, 0x0
10450	1afffff5	bne 1042c <inloop>

Edited output For output Created by the command
`%objdump -d a.out`
`%objdump -d -S a.out adds source code`

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

Arithmetic and
logic

Data Movement

Control Flow

Miscellaneous

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  
register direct "addressing"
```

register r1



register r0

```
mov  r0, 100
```

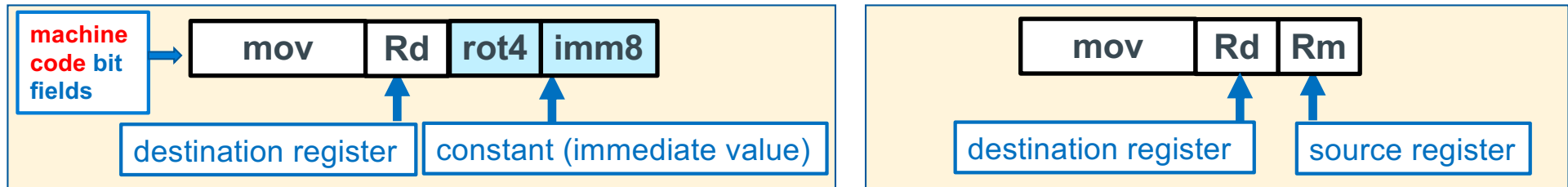
```
// Expands an imm8 value 100  
// stored in the instruction  
// into the register r0  
Immediate "addressing"
```

100



register r0

mov – Copies Register Content between registers



	4	3	4	1	4	4	8	4
R type:	1110	000	Opcode	S	0000	Rd	00000000	Rm
I type:	1110	001	Opcode	S	0000	Rd	Rotate	Imm8

↑
 { 1101 - MOV
 1111 - MVN

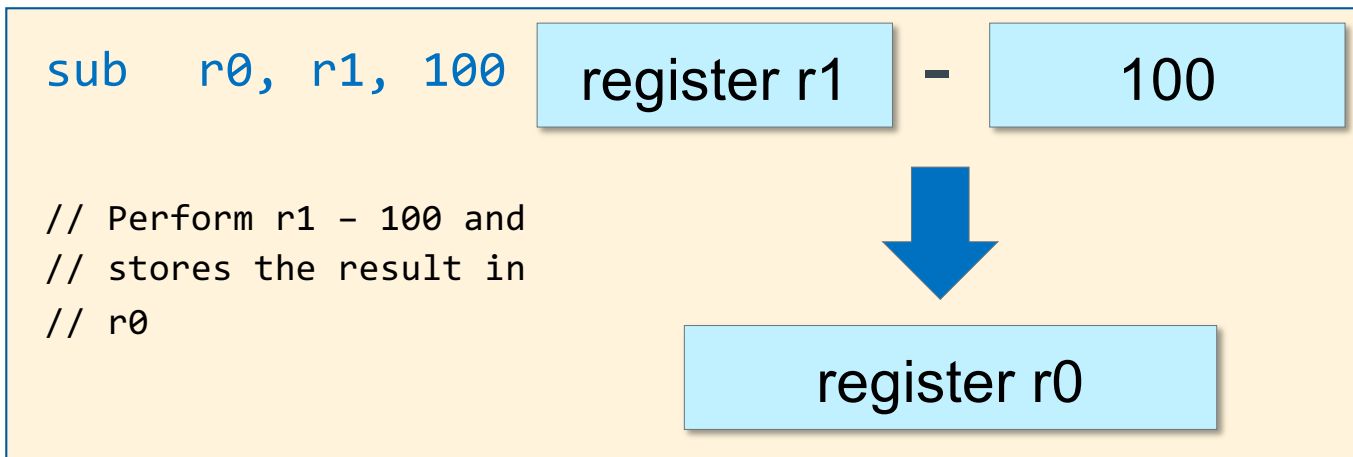
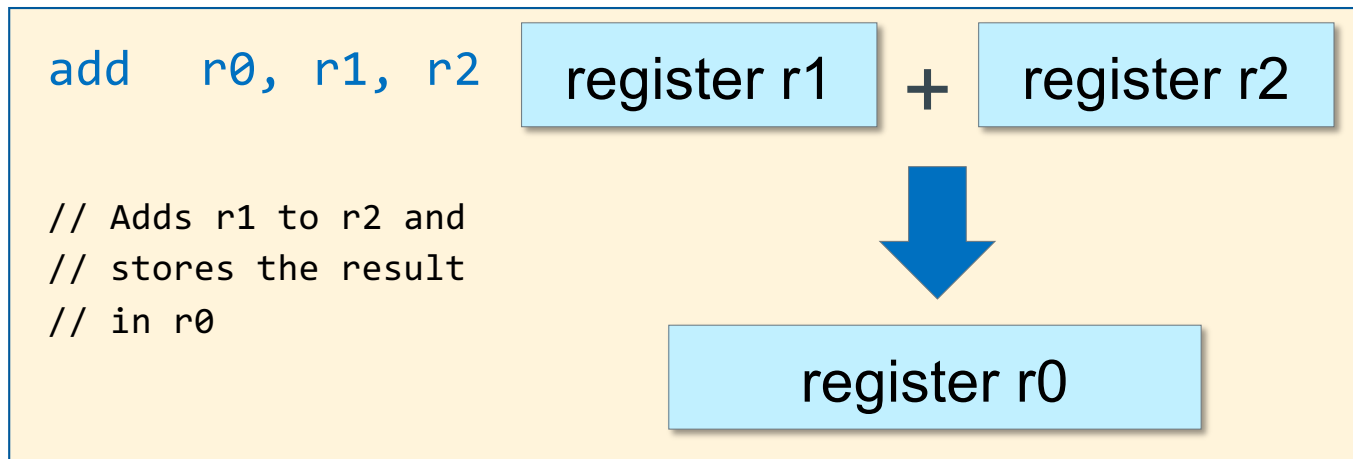
```

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

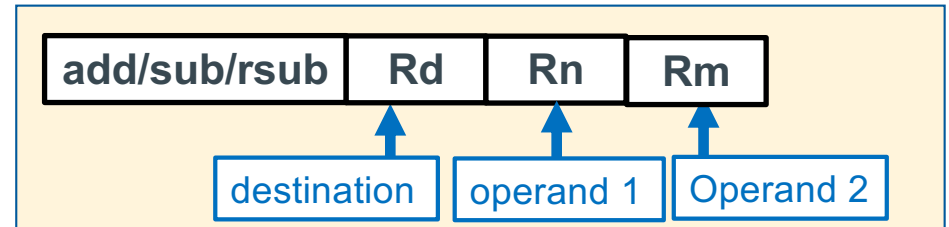
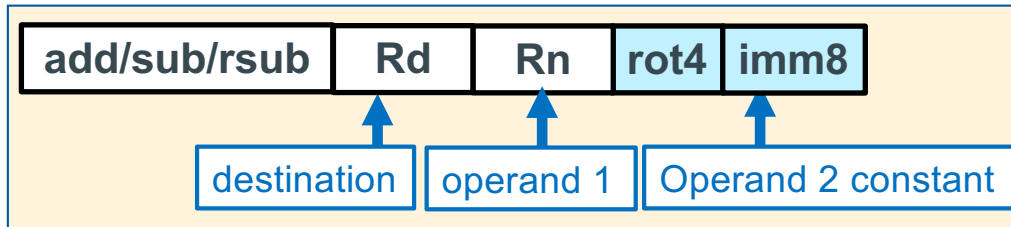
```

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

First Look: Add/Sub Registers



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
    
```

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
```

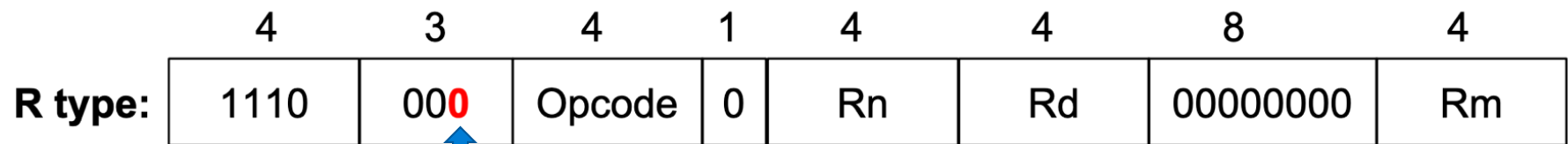
```
a = (b + c) - 5;
r0 = (r1 + r2) - 5;
```

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


R (register) Type Data Processing: Machine Code

- 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

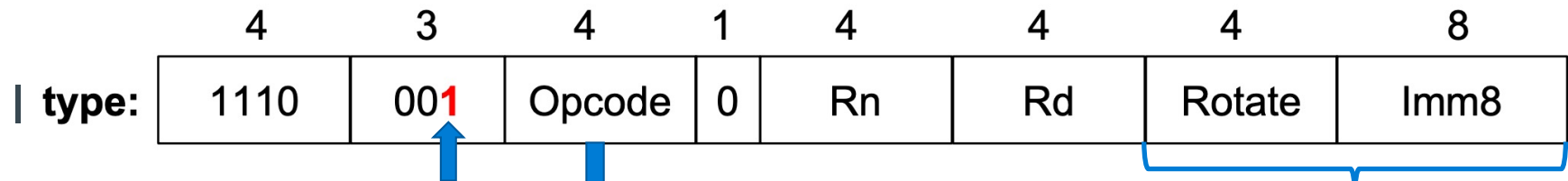
0xe0810003

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

I (immediate) Type Data Processing: Machine Code

- 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

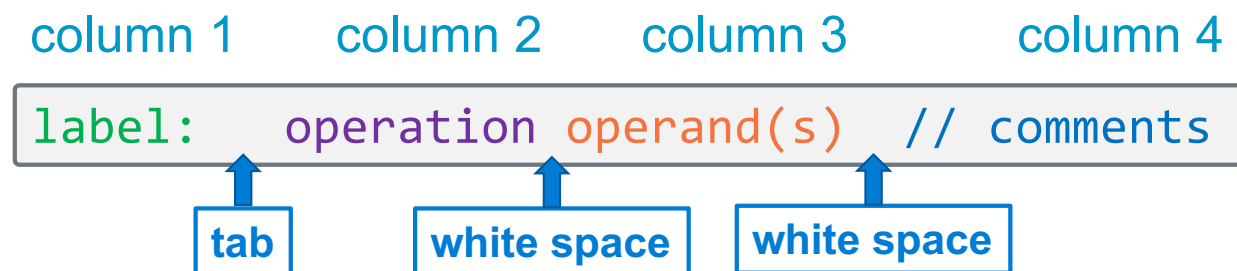
1110 0000 1000 0001 0000 0000 0011 0001

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

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**

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

1. **Labels** (optional); starts in column 1

- **Only put a label on a line** when you need to **associate** a name (a global variable, a function name, a loop/ branch target, etc.) to that **lines** location in memory
- You then refer to the address **by name** in an **instruction**

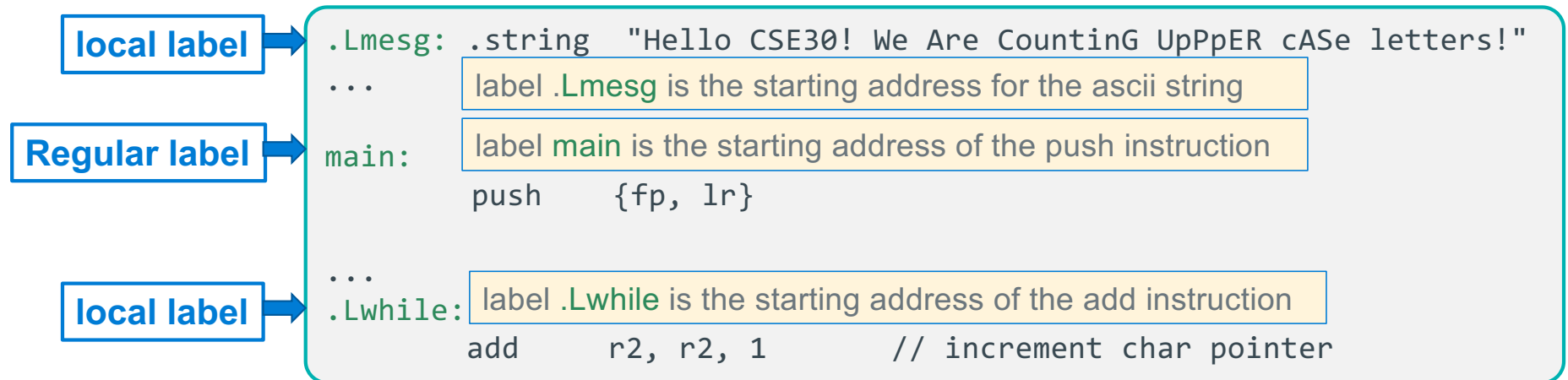
2. **Operation type 1: assembler directives** (all start with a period e.g. **.word**)

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 //

Labels in Arm Assembly



- Remember, a **Label** associates a **name** with **memory location**
- **Regular Label:**
 - Used with a **Function name** (label) or all **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**
 - a) branches: if switch, goto, break, continue,
 - b) loops: for, while, do-while
 2. **Anonymous variables** (the address of **string** not the address of **foo** in the following)
`char *foo = "anonymous variable"`

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

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

Example: Assembler Directive and Instructions

assembler directive `.equ` does not allocate any memory (NULL = 0)

Regular label `main` is associated with memory location 0x3000

Local label `.Lloop` is associated with memory location 0x3004

space.S

```
10  .equ NULL, 0
11 main:
12 3000 0310A0E1      mov     r1, r3
13 .Lloop:
14 3004 043083E2      add     r3, r3, 4
15 3008 001093E5      ldr     r1, [r3]
16 300c 000051E3      cmp     r1, NULL
17 3010 FBFFFF1A      bne     .Lloop
```

output generated with
`gcc -c -Wa,-ahlns space.S`
partial output is shown

Memory Contents

Warning contents shown in "reverse" byte order: Lsb – Msb

Instruction Memory Addresses (lowest 2-bits are always are 00)
Notice alignment and how addresses increase by 4 (32-bit instructions)

Unconditional Branching – Forces Execution to Continue at a Specified Label (goto)



Unconditional Branch instruction (*branch to only local labels in CSE30*)

b **.Llabel**

- Causes an unconditional branch (aka goto) to the instruction with the address **.Llabel**
- **.Llabel** is called a **branch target label** (the "*target*" of a branch instruction)
- **Be careful! do not to branch to a function label!**
- **.Llabel**: pc is the base register with the offset being **imm24** shifted left two bits (+/- 32 MB)
 - **imm24** is the **number of instructions** from **pc+8**

```
        b        .Ldone
        :
.Ldone:  add      r0, EXIT_SUCCESS    // set return value
```

Examples of of Unconditional Branching

Unconditional Branch Forward

```
b .Lforward
add r1, r2, 4
add r0, r6, 2
add r3, r7, 4
.Lforward:
sub r1, r2, 4
```

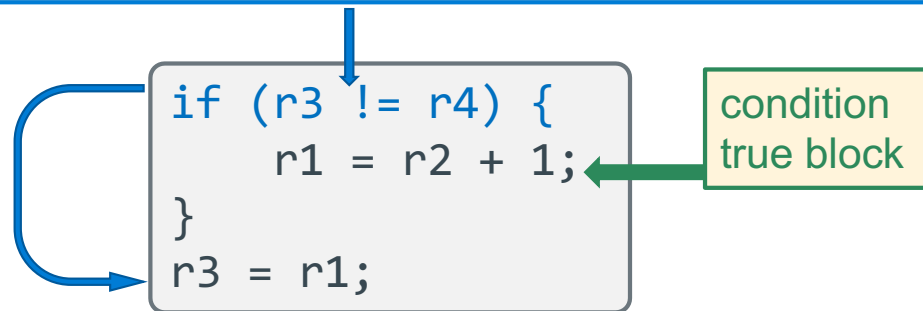
Backward Branch (Infinite loop)

```
.Lbackward:
    add r1, r2, 4
    sub r1, r2, 4
    add r4, r6, r7
    b .Lbackward
    // not reachable unless there is a label
    after the .b above
```

- Branches are used to change execution flow using labels as the branch target
- In these example, **.Lforward** and **.Lbackward** are the branch target labels
- Branch target labels are placed at the beginning of the line (or above it)
- Caution: Backward branches should only used with loops!

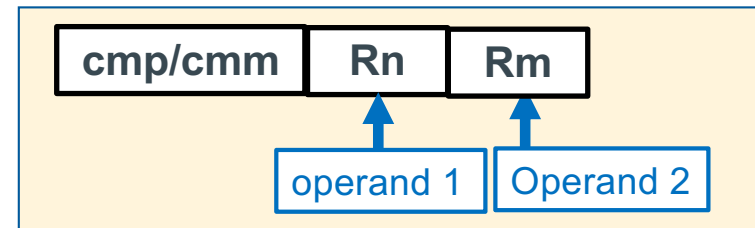
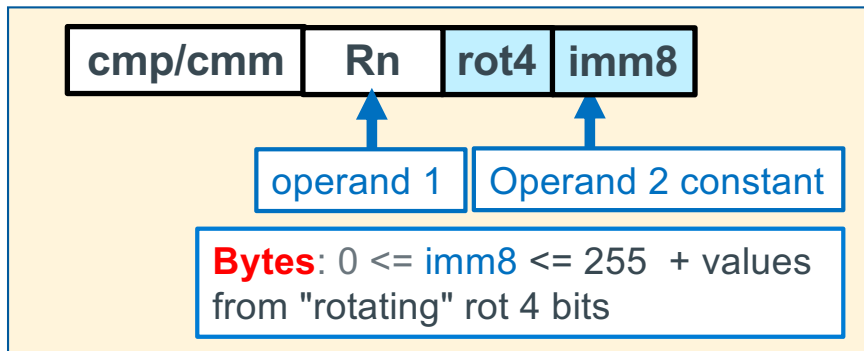
Anatomy of a Sample Branch: Changing the Next Instruction to Execute

Branch condition depend on the result of a Test (comparison)
(This test is called a **branch guard**)



- **Branch guard**: determines whether to execute the "true" block
- Step 1: evaluate the branch guard(s) (involves one or more compares/tests)
- Step 2: If **branch guard** evaluates to false
 - then **branch around** the **true block**
 - else execute the true block

cmp/cmm – Making Conditional Tests



```
cmp  Rn, constant    // Rn - constant; then sets condition flags
cmm  Rn, constant    // Rn + constant; then sets condition flags
cmp  Rn, Rm           // Rn - Rm; then sets condition flags
cmm  Rn, Rm           // Rn + Rm; then sets condition flags
```

The values stored in the registers `Rn` and `Rm` are not changed
The assembler will automatically substitute `cmm` for negative immediate values

```
cmp    r1, 0          // r1 - 0 and sets flags on the result
cmp    r1, r2          // r1 - r2 and sets flags on the result
```


Quick Overview of the Condition Bits/Flags

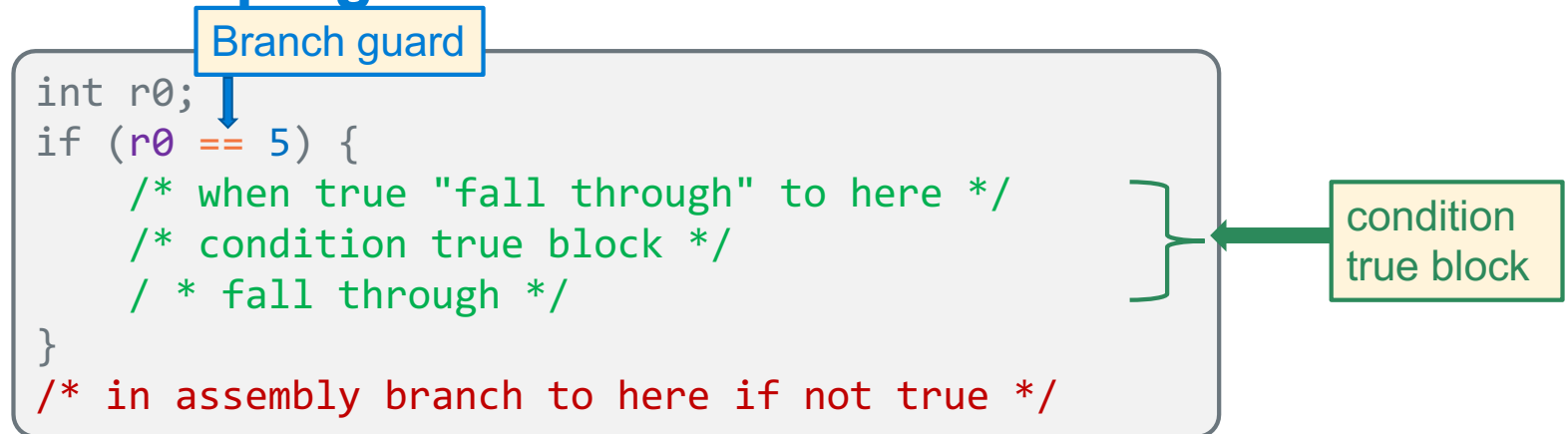


- The CPSR is a special register (like the other registers) in the CPU
- The four bits at the left are called the Condition Code flags
 - Summarize the result of a previous instruction
 - Not all instruction will change the CC bits
- Specifically, Condition Code flags are set by cmm/cmp (and others)

Example: `cmp r4, r3`


- **N** (Negative) flag: Set to 1 when the result of $r4 - r3$ is negative, set to 0 otherwise
- **Z** (Zero) flag: Set to 1 when the results of $r4 - r3$ is 0, set to 0 otherwise
- **C** (Carry bit) flag: Set to 1 when $r4 - r3$ does not have a borrow, set to 0 otherwise
- **V** flag (oVerflow): Set to 1 when $r4 - r3$ causes an overflow, set to 0 otherwise

Program Flow: Keeping the same "*Block Order*" as C



- In ARM32, you either **fall through** (execute the *next instruction in sequence*) or **branch around to a specific instruction** and then *resume* sequential instruction execution
- In order to keep the **same block order** as the **C version** that says: **fall through** to the **condition true** block when the **branch guard** evaluates to be **true**
 - Assembly: **invert** the **condition test** to **branch around** the **condition true** block
- **Summary:** In ARM32 use a **condition test** that **specifies the opposite** of the condition used in C , then **branch around** the **condition true** block

Conditional Branch: Changing the Next Instruction to Execute



```
cmp    r3, r4
beq    .Ldone
// otherwise fall through do the add
add    r1, r2, 1
// now fall through to mov
.Ldone:
mov    r3, r1
```

```
if (r3 != r4)
    r1 = r2 + 1;
r3 = r1;
```

Condition	Meaning	Flag Checked
BEQ	Equal	Z = 1
B	Always (unconditional)	

```
cmp    r3, r4 // r3 - r4
// if r3 != r4 sets Z = 0
```

1. Use a **cmp/cmm** instruction to set the condition bits
2. Follow the **cmp/cmm** with one or more variants of the conditional branch instruction **Conditional branch instructions** if evaluate to true (bases on the CC bits set) will go to the instruction with the branch label. Otherwise, it executes the instruction that follows
 - You can have one or more conditional branches after a single cmp/cmm

Examples: Guards (Conditional Tests) and their Inverse

Compare in C	<i>"Inverse"</i> Compare in C
==	!=
!=	==
>	<=
>=	<
<	>=
<=	>

- Changing the conditional test (guard) to its inverse, allows you to swap the order of the blocks in an if else statement

Conditional Branch: Changing the Next Instruction to Execute



Branch instruction

bsuffix .Llabel

- Bits in the condition field specify the **conditions** when the branch happens
- If the condition evaluates to be **true**, the **next instruction executed is located at .Llabel:**
- If the condition evaluates to be **false**, the **next instruction executed** is located immediately after the branch
- **Unconditional branch** is when the condition is **"always"**

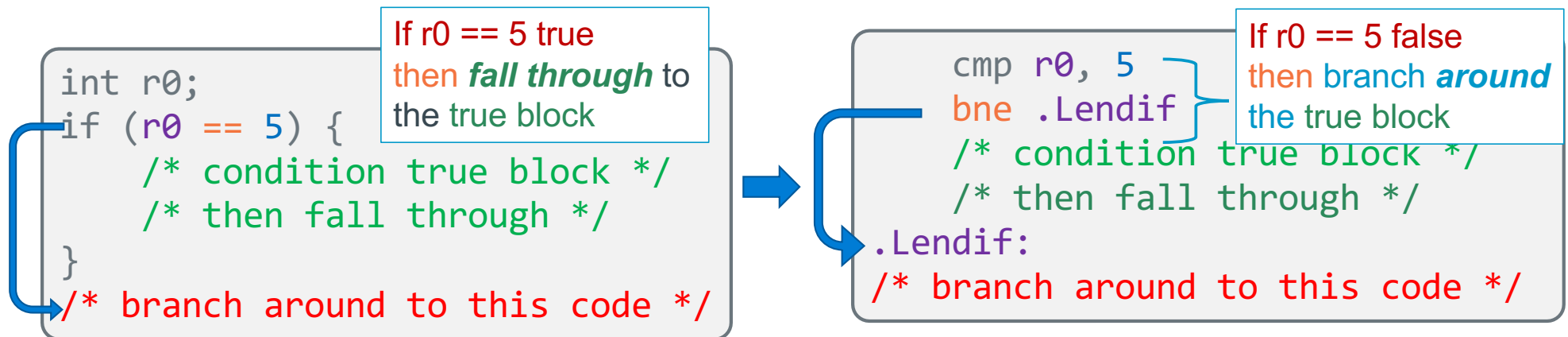
Condition	Meaning	Flag Checked
BEQ	Equal	Z = 1
BNE	Not equal	Z = 0
BGE	Signed \geq ("Greater than or Equal")	N = V
BLT	Signed $<$ ("Less Than")	N \neq V
BGT	Signed $>$ ("Greater Than")	Z = 0 && N = V
BLE	Signed \leq ("Less than or Equal")	Z = 1 N \neq V
BHS	Unsigned \geq ("Higher or Same") or Carry Set	C = 1
BLO	Unsigned $<$ ("Lower") or Carry Clear	C = 0
BHI	Unsigned $>$ ("Higher")	C = 1 && Z = 0
BLS	Unsigned \leq ("Lower or Same")	C = 0 Z = 1
BMI	Minus/negative	N = 1
BPL	Plus - positive or zero (non-negative)	N = 0
BVS	Overflow	V = 1
BVC	No overflow	V = 0
B (BAL)	Always (unconditional)	

Program Flow: Simple If statement, No Else

Approach: **adjust** the conditional test then **branch around** the **true block**

Use a **conditional test** that specifies the **inverse** of the condition used in C

C source Code	Incorrect Assembly	Correct Assembly
<pre>int r0; if (r0 > 10)</pre>	<pre>cmp r0, 10 bgt .Lendif .Lendif:</pre>	<pre>cmp r0, 10 ble .Lendif .Lendif:</pre>



Branch Guard "*Adjustment*" Table

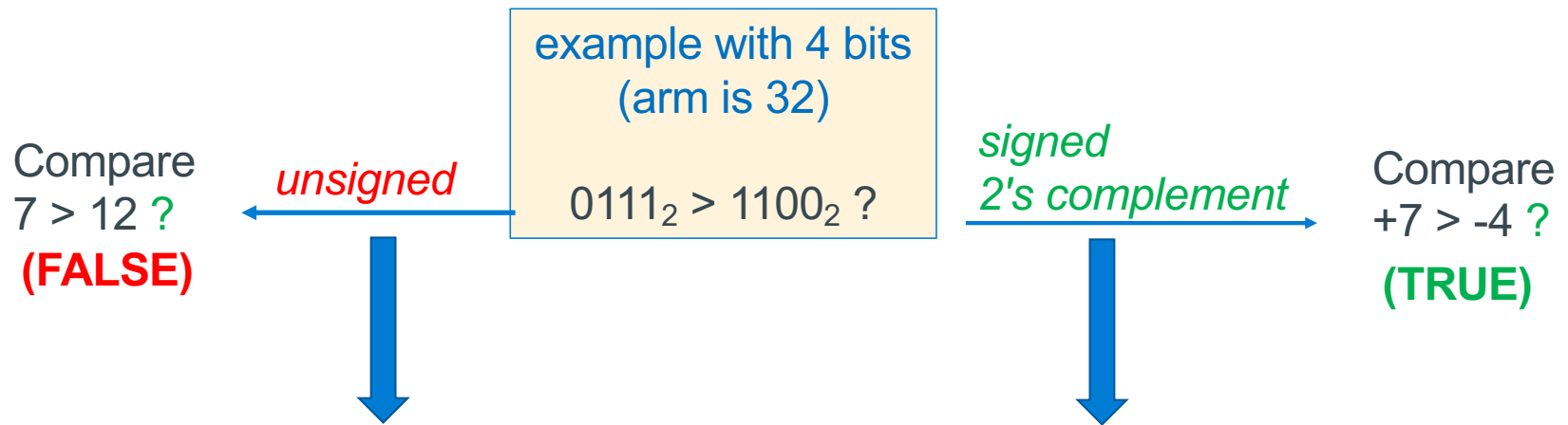
Preserving Block Order In Code

Compare in C	"Inverse" Compare in C	"Inverse" Signed Assembly	"Inverse" Unsigned Assembly
==	!=	bne	bne
!=	==	beq	beq
>	<=	ble	bls
>=	<	blt	blo
<	>=	bge	bhs
<=	>	bgt	bhi

```
if (r0 compare 5)
    /* condition true block */
    /* then fall through */
}
```

```
cmp r0, 5
inverse .Lelse
    // condition true block
    // then fall through
.Lendif:
```

When do you use a Signed or Unsigned Conditional Branch?



Condition	Suffix For Unsigned Operands:	Suffix For Signed Operands:
>	BHI (<i>Higher Than</i>)	BGT (<i>Greater Than</i>)
>=	BHS (<i>Higher Than or Same</i>) (<i>BCS</i>)	BGE (<i>Greater Than or Equal</i>)
<	BLO (<i>Lower Than</i>) (<i>BCC</i>)	BLT (<i>Less Than</i>)
<=	BLS (<i>Lower Than or Same</i>)	BLE (<i>Less Than or Equal</i>)
==	BEQ (<i>Equal</i>)	
!=	BNE (<i>Not Equal</i>)	

Anatomy of a Conditional Branch: If statement

Branch condition
Test (branch guard)

```
if (r0 == 5) {  
    /* condition block #1 */  
} else {  
    /* condition block #2 */  
    /* fall through */  
}
```

condition
true block

condition
false block

- In **C**, when the branch guard (condition test) evaluates **non-zero** you **fall through** to the **condition true** block, otherwise you branch to the **condition false** block
- Block order: (the **order** the **blocks appear** in C code) can be changed by **inverting** the conditional test, **swapping** the order of the **true** and **false** blocks

Branch condition
Test (branch guard)

```
if (r0 != 5) {  
    /* condition block #2 */  
} else {  
    /* condition block #1 */  
    /* fall through */  
}
```

condition
true block

condition
false block