

Version 1.03

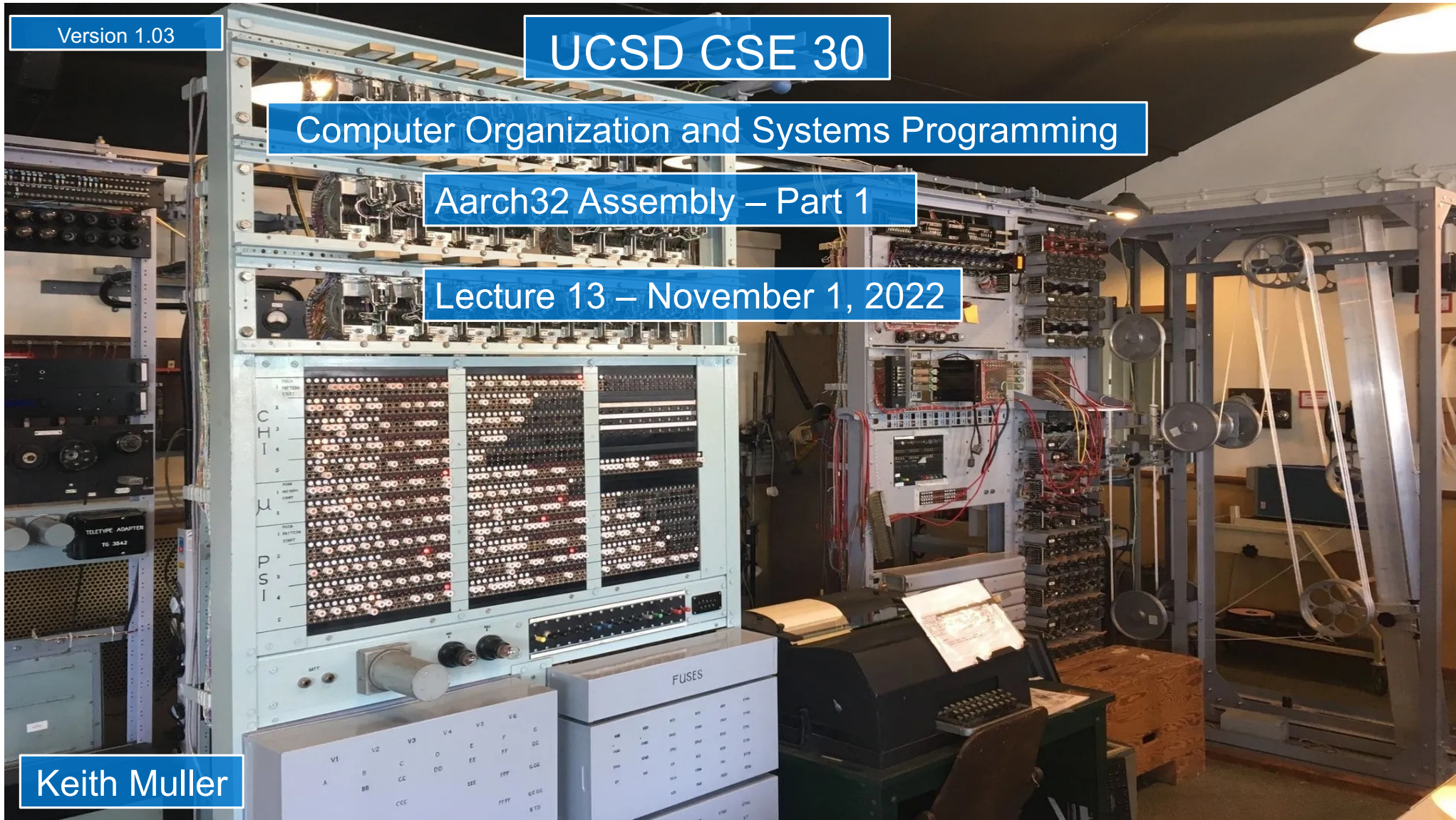
UCSD CSE 30

Computer Organization and Systems Programming

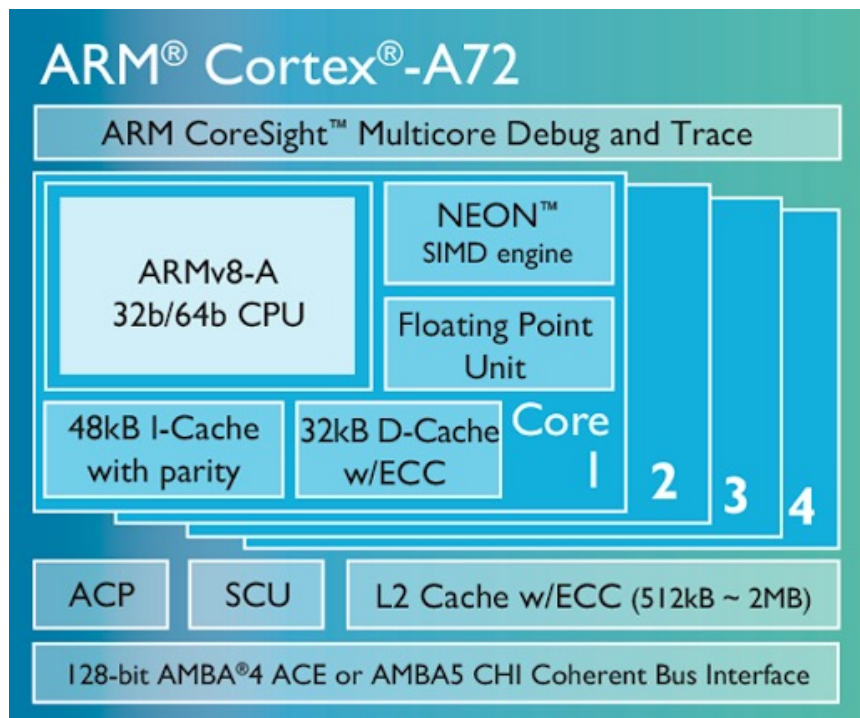
Aarch32 Assembly – Part 1

Lecture 13 – November 1, 2022

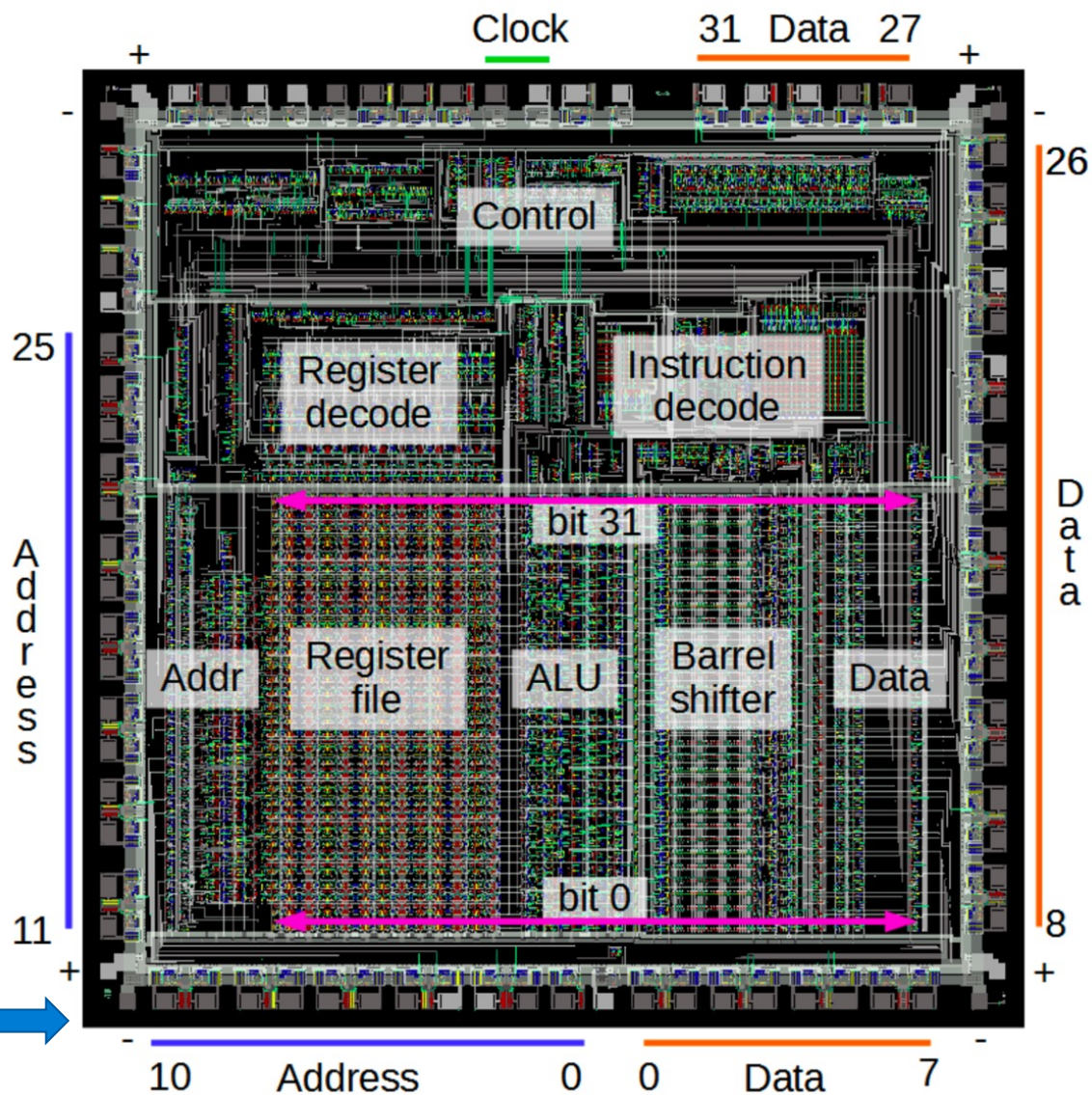
Keith Muller



Arm Core Organization & Floorplan Examples



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



Memory Triangle: Hardware Cost/Performance/Capacity Tiers

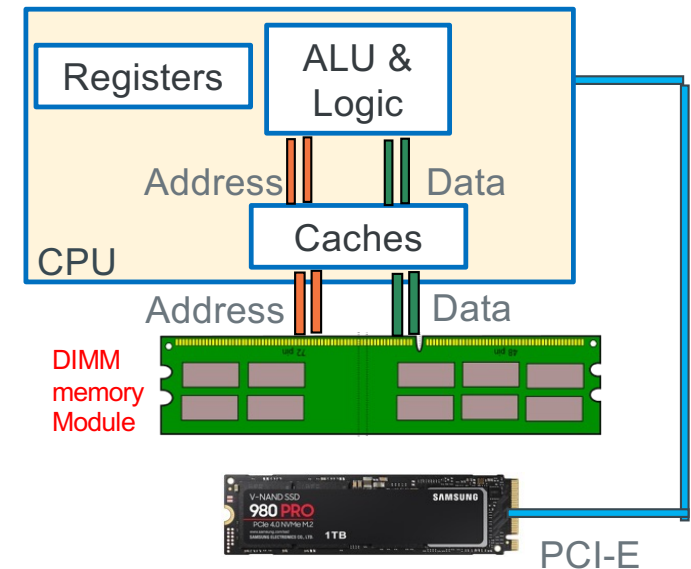
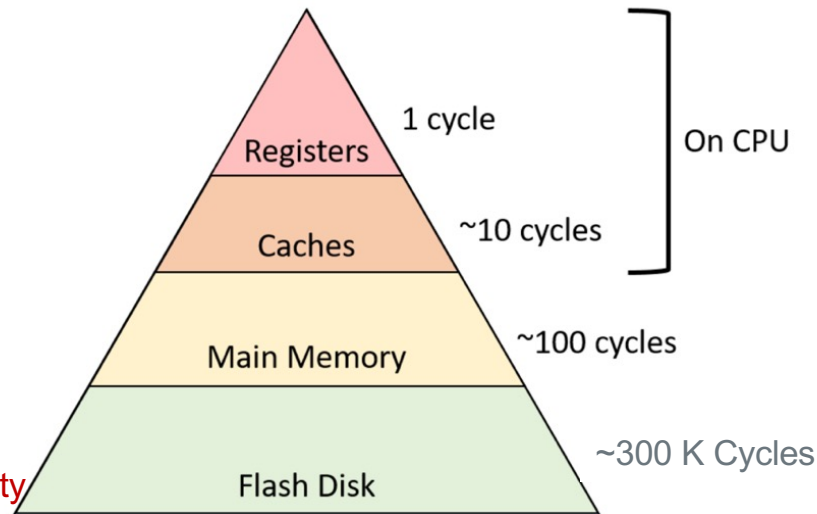
Goal: keep most Data
Accesses high in the
Triangle

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



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

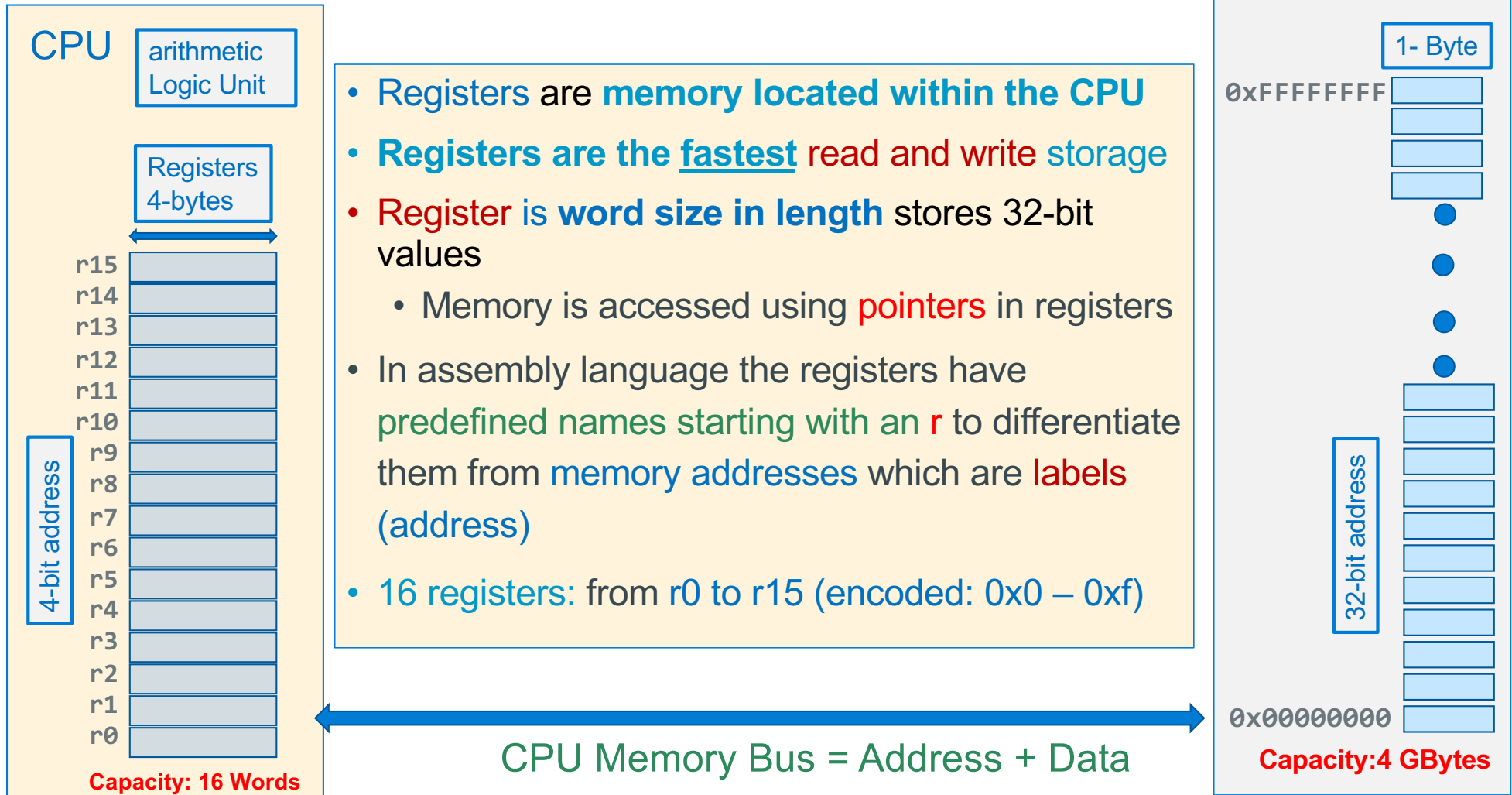
Assume 1 clock cycle
per machine instruction



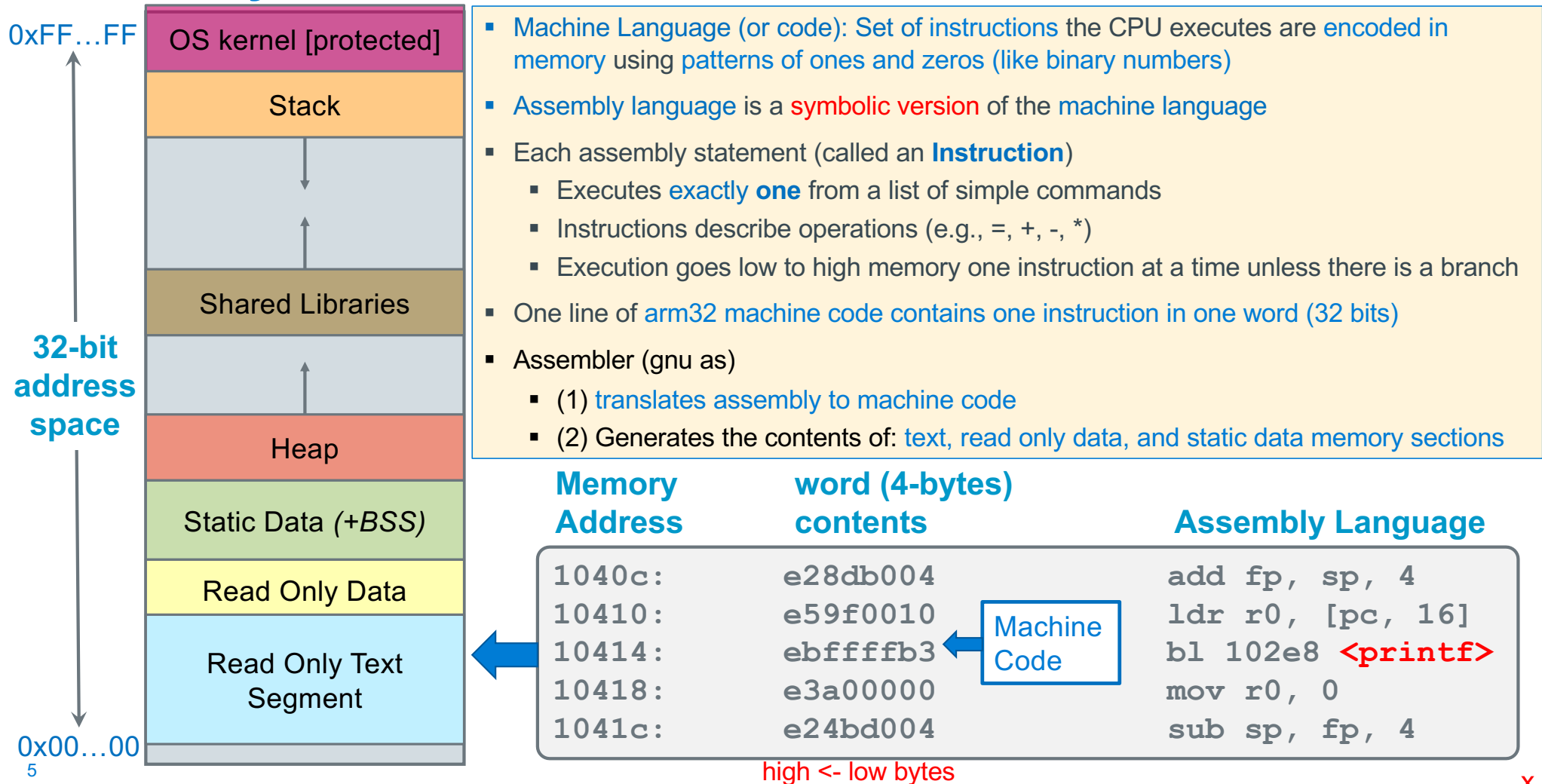
Clock cycle \approx time to access; larger is slower

Design Tradeoff: Based on workload considering cost and performance targets

32-Bit Arm - Registers

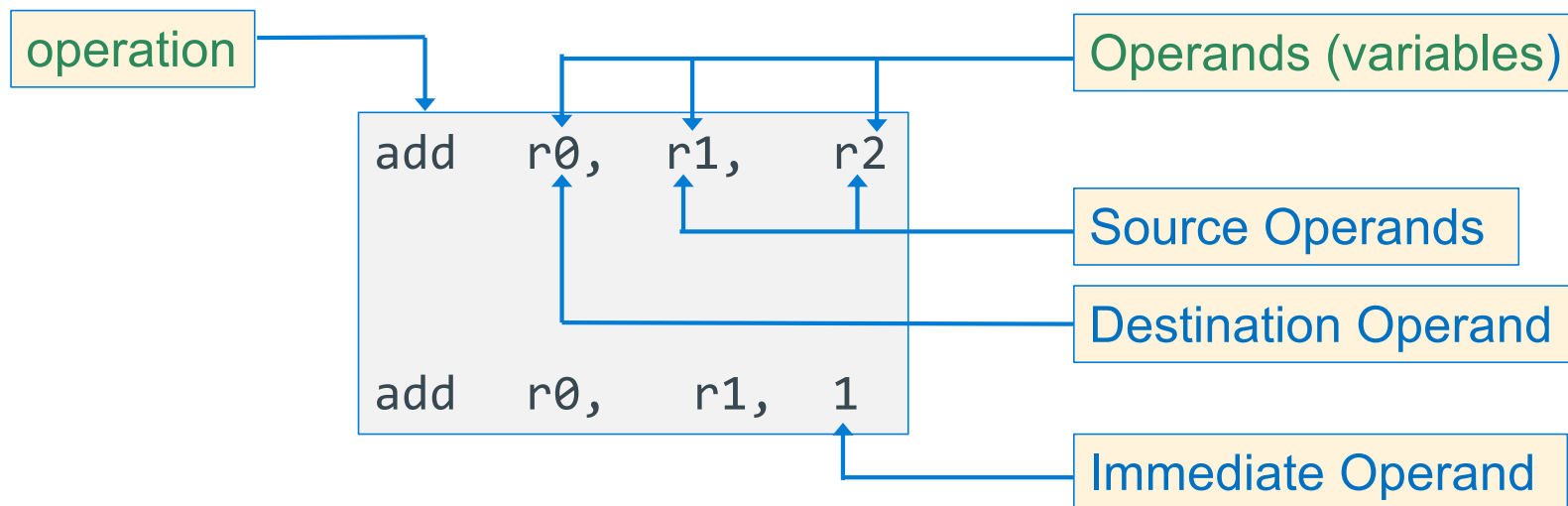


Assembly and Machine Code



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


32-Bit Arm - Registers

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

Arithmetic & Logic Unit (ALU)

registers

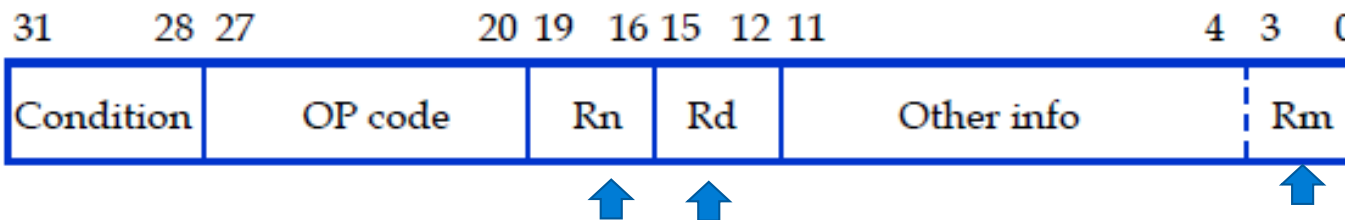
4-bytes

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

4-bit address

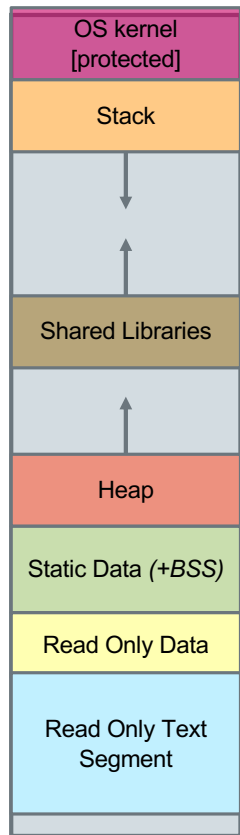
Capacity: 16 Words

- 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)



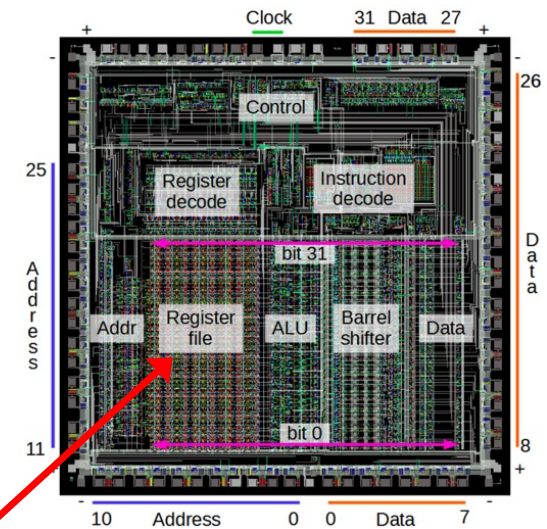
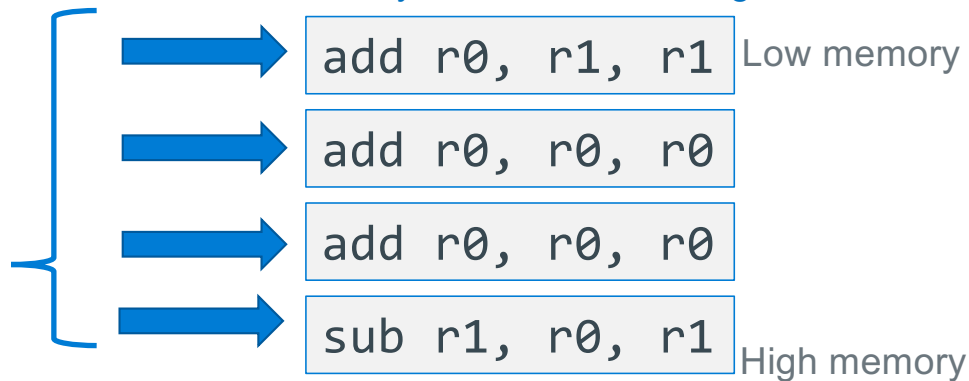
Program Execution: A Series of Instructions

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

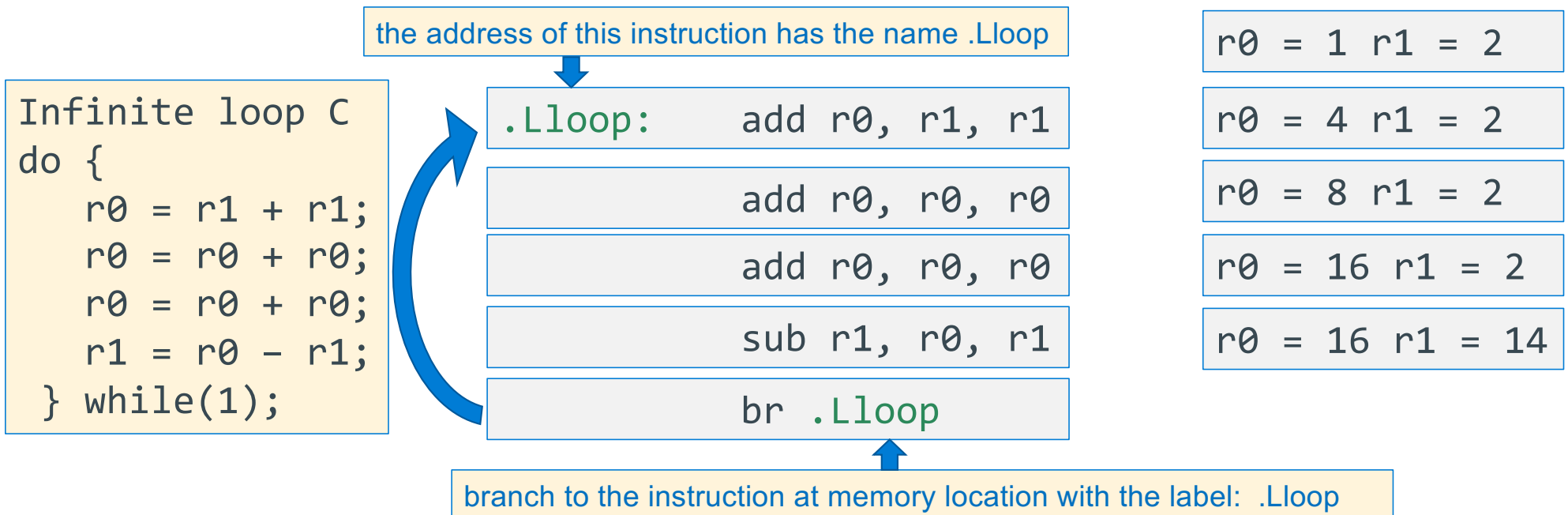
r0 = 8 r1 = 2

r0 = 16 r1 = 2

r0 = 16 r1 = 14

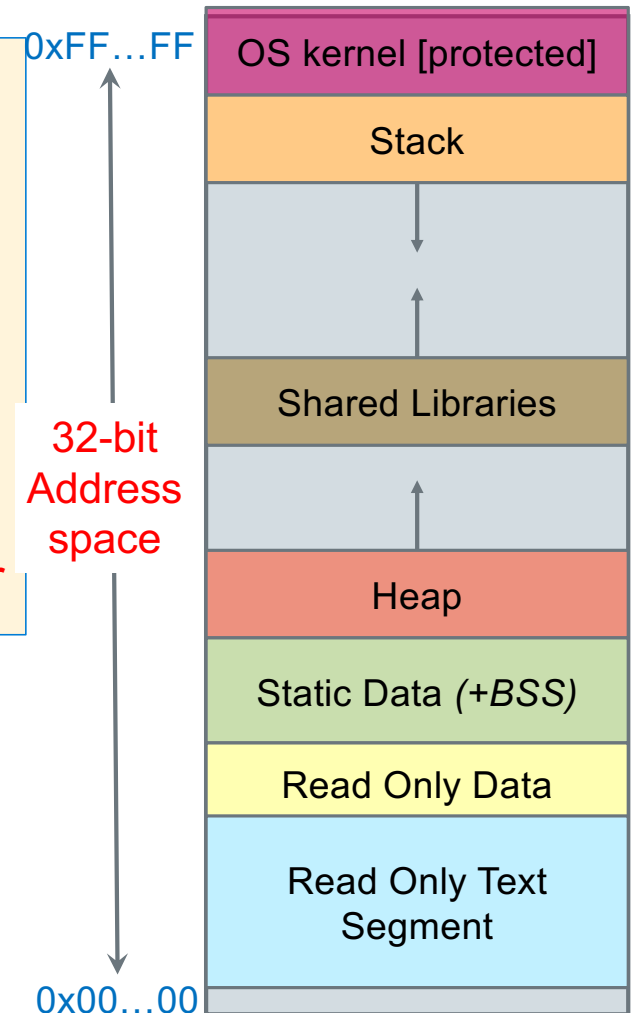
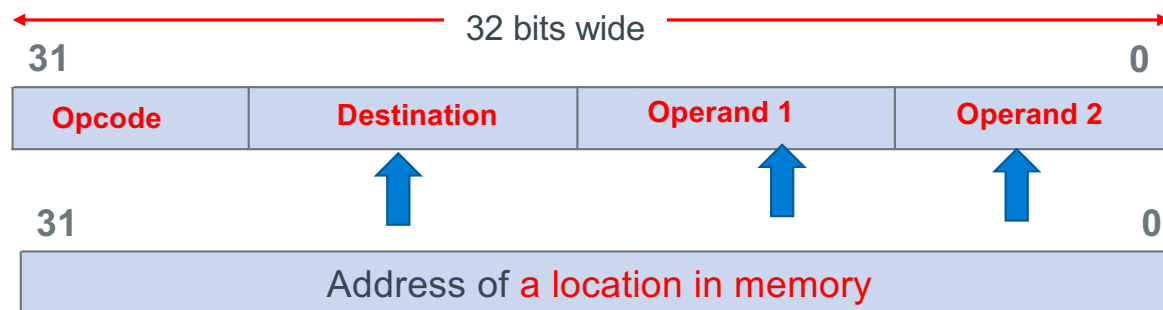
Program Execution: Looping in the Execution Flow

- 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")



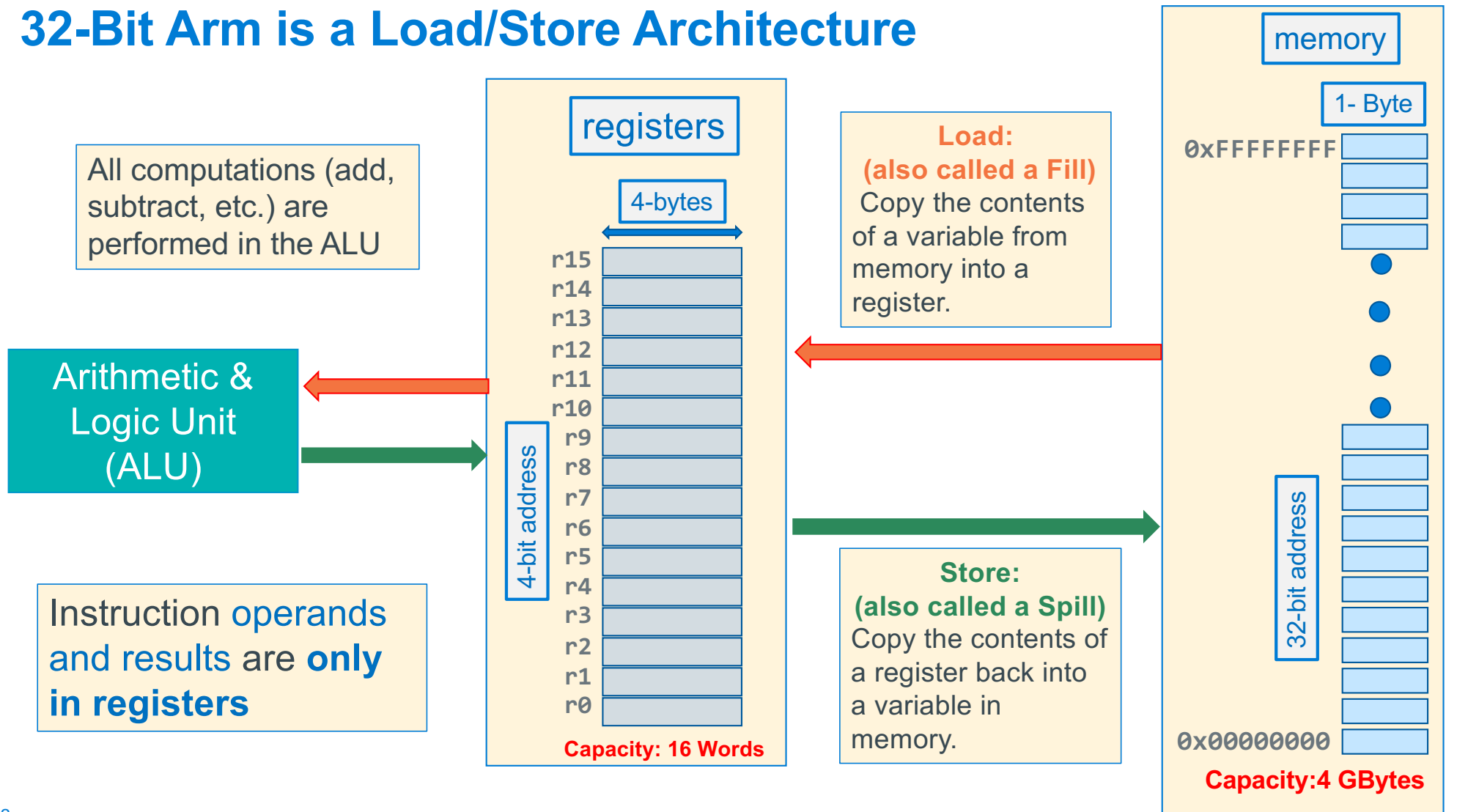
How to Access Memory?

- 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



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
```

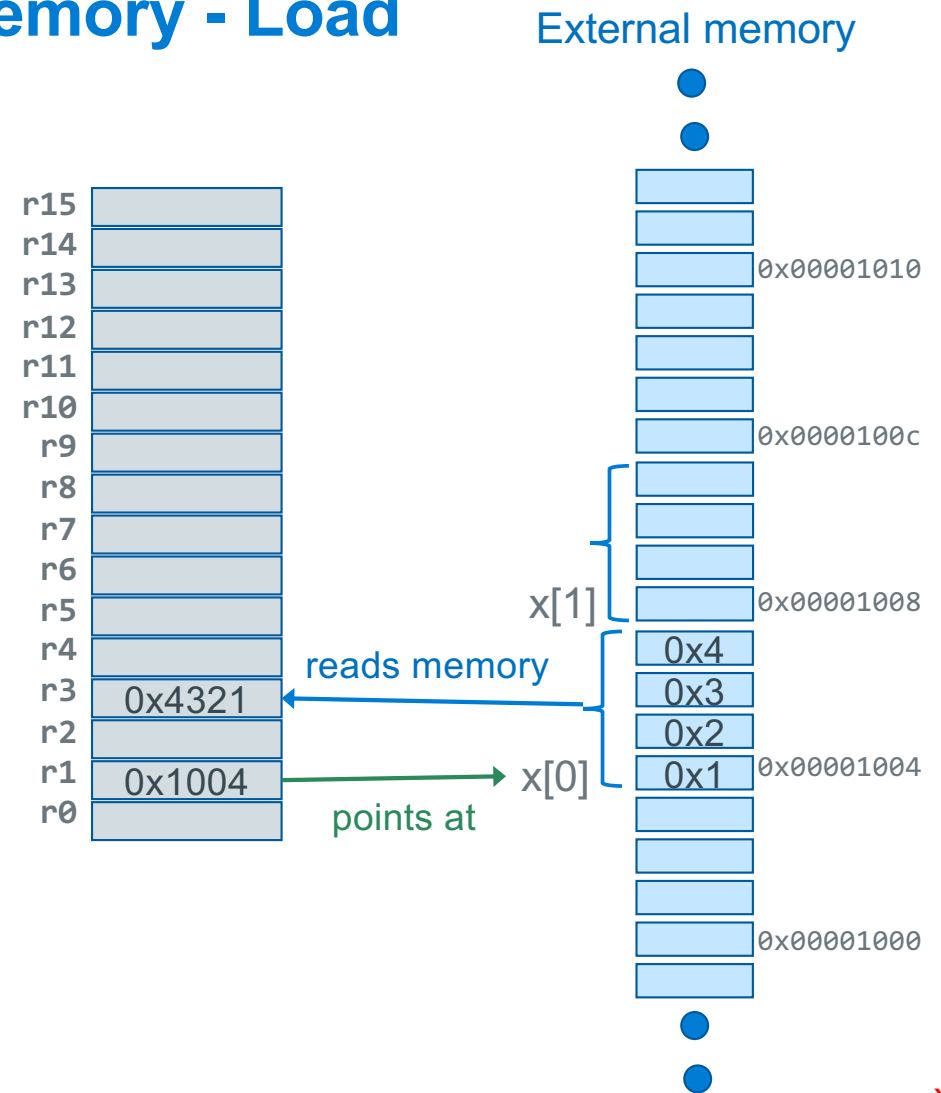
Load register from memory (read)

```
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
```

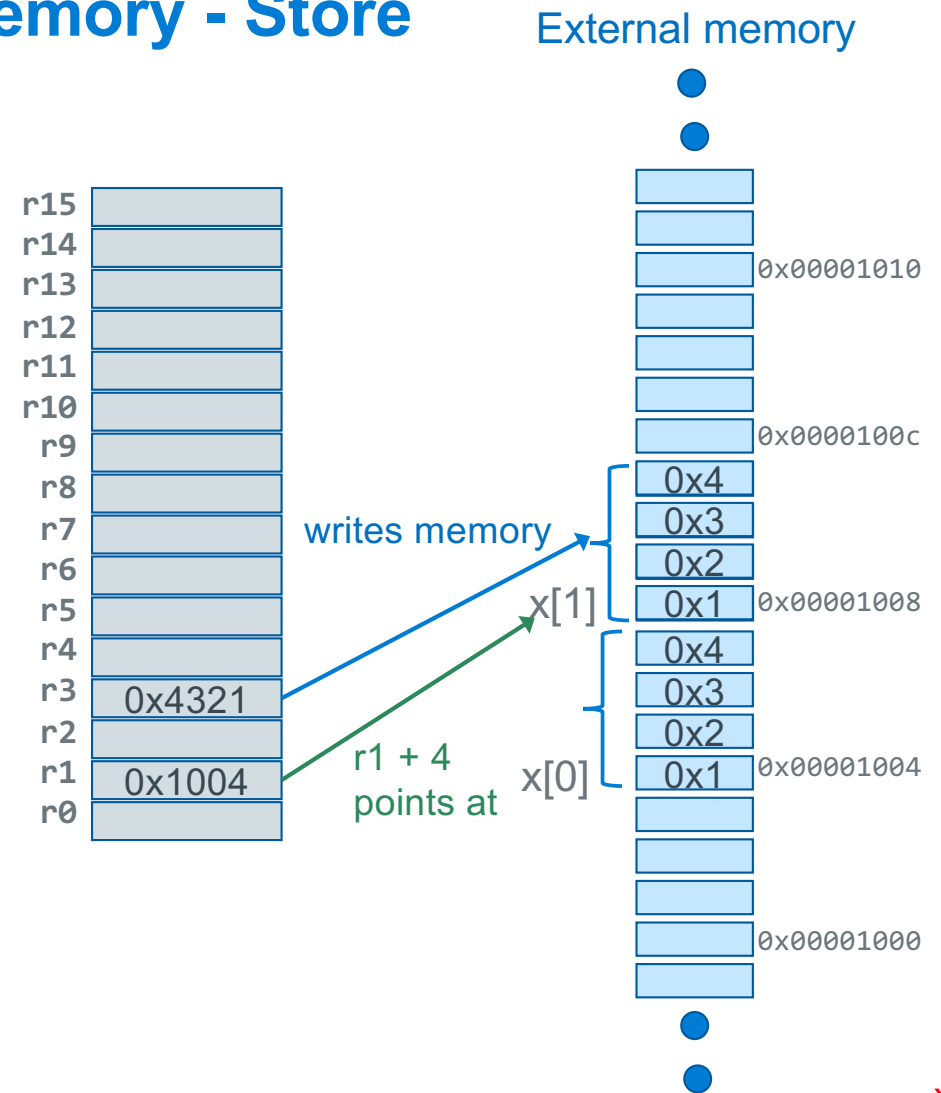
Store register **to** memory (write)

```
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 1.03

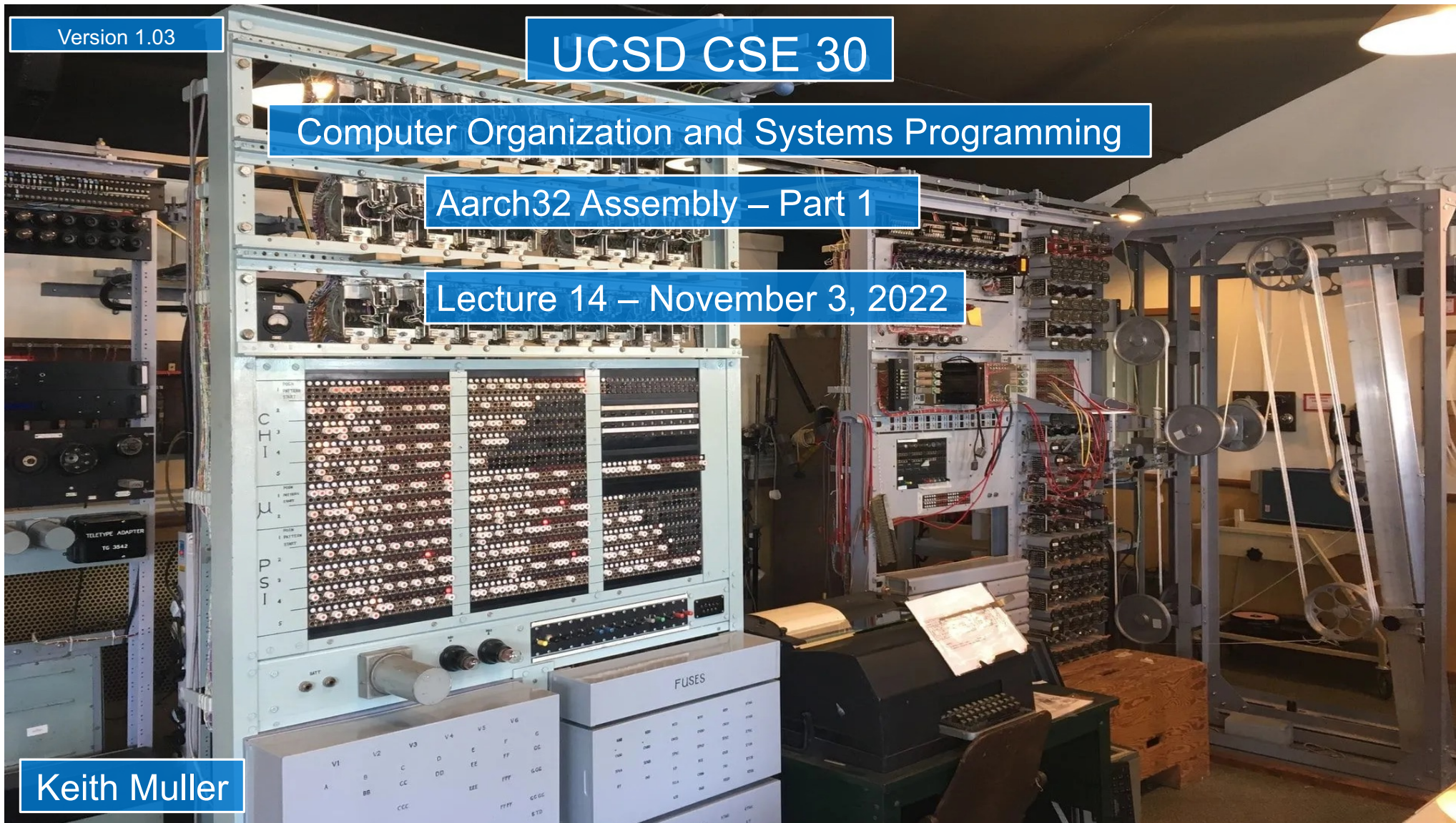
UCSD CSE 30

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
```

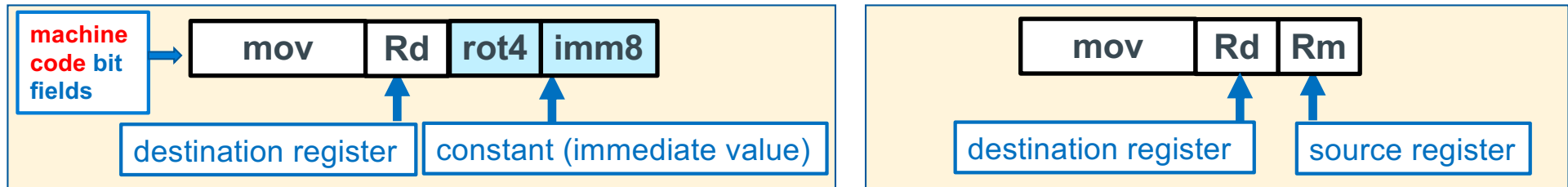
```
// Expands an imm8 value 100  
// stored in the instruction  
// into the register r0  
Immediate "addressing"
```

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

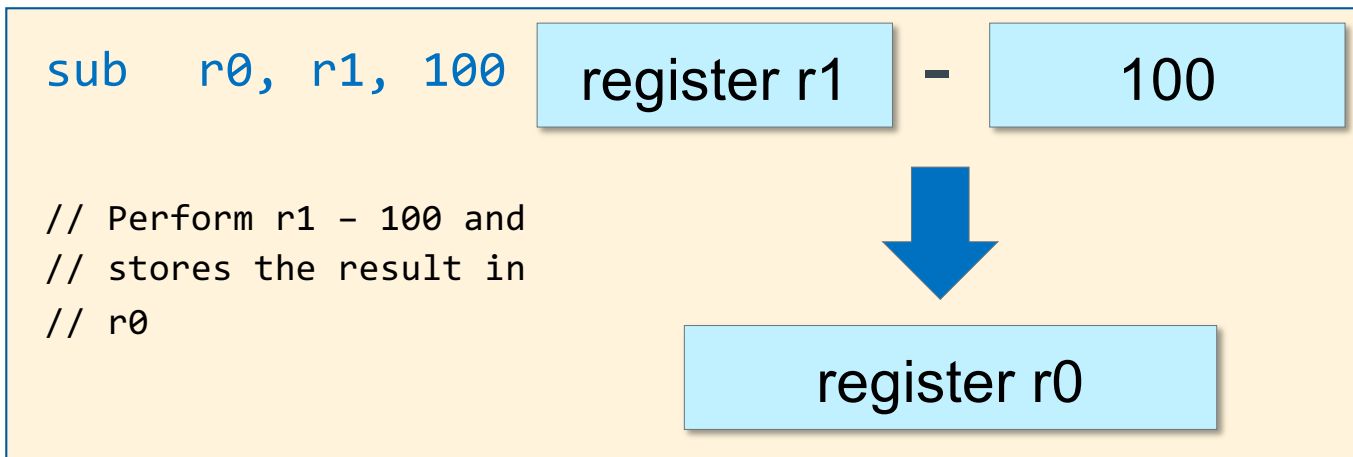
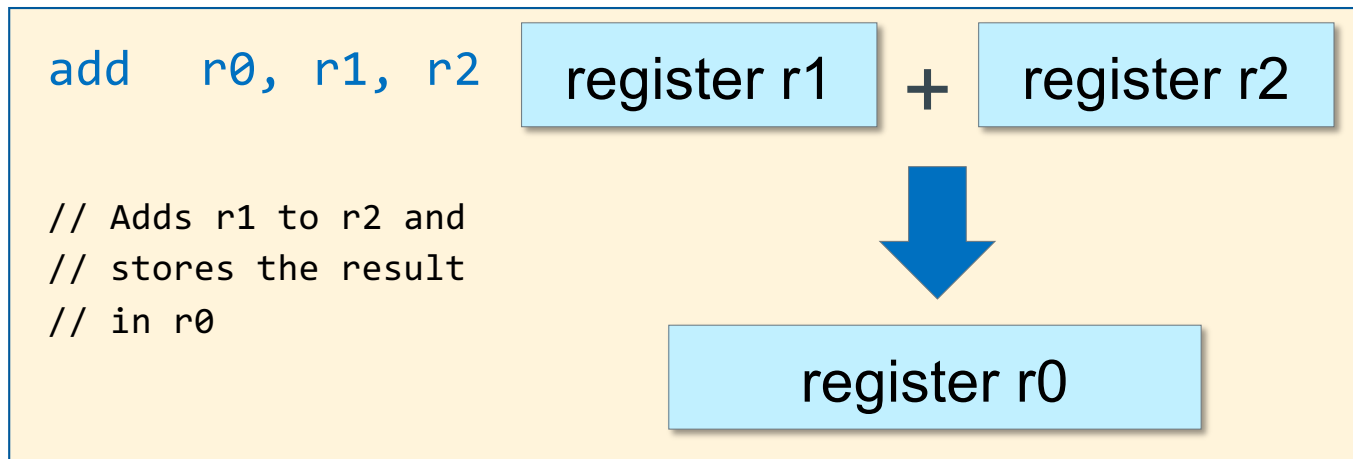
```

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

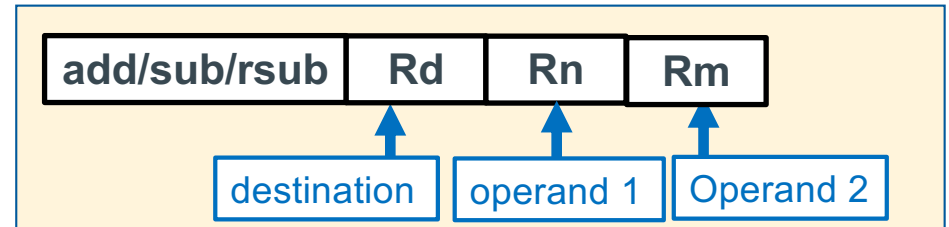
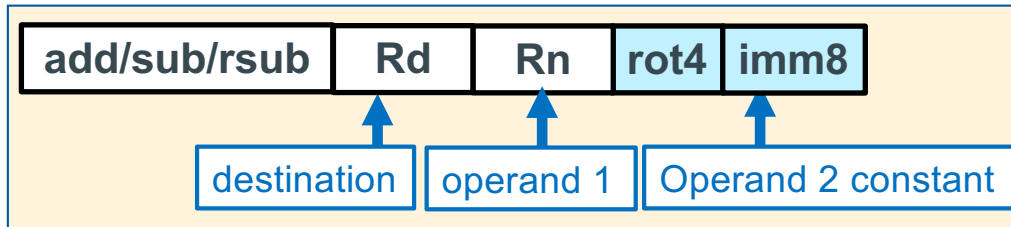
```

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
```

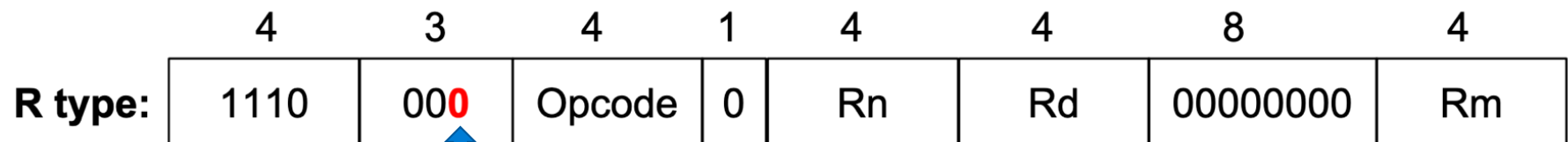
```
a = (b + c) - 5;
r0 = (r1 + r2) - 5;
```

```
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

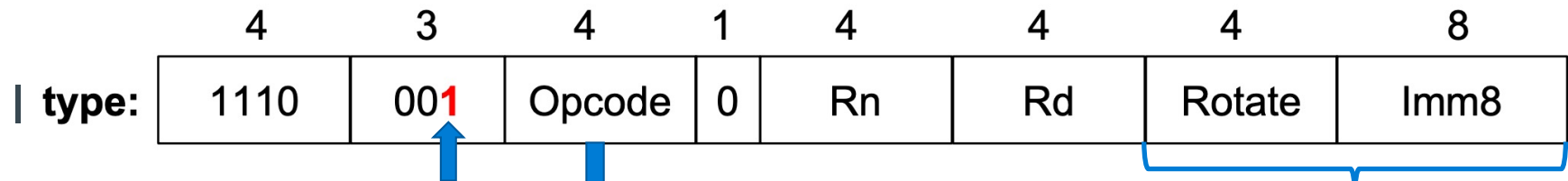
0xe0810003

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

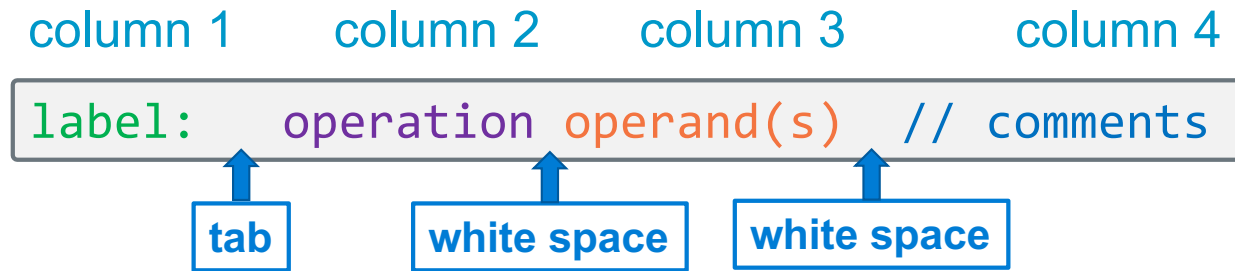
1110 0000 1000 0001 0000 0000 0011 0001

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 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**

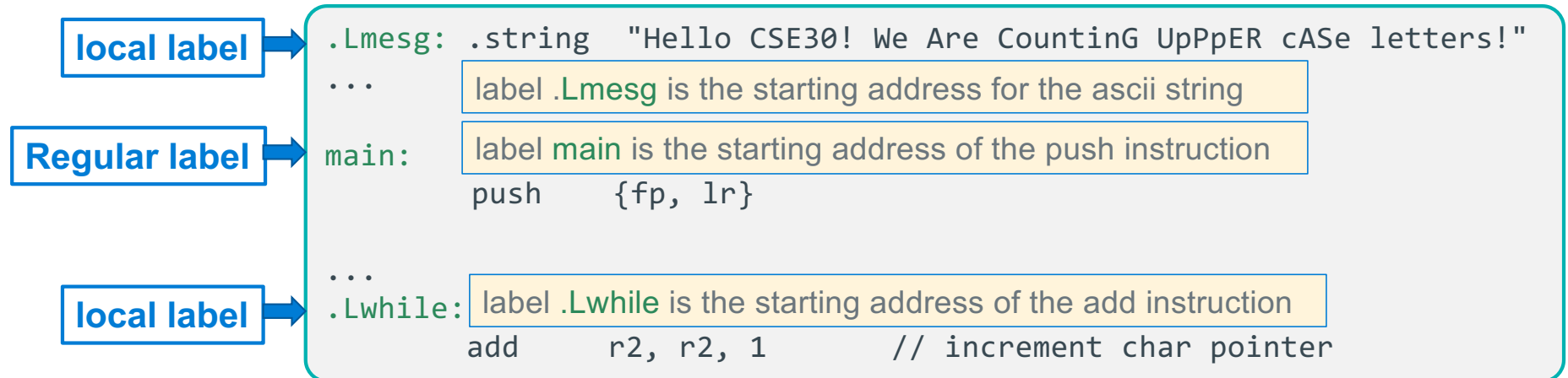
2. **Operation type 1: assembler directives** (all start with a period e.g. **.word**)

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 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
```

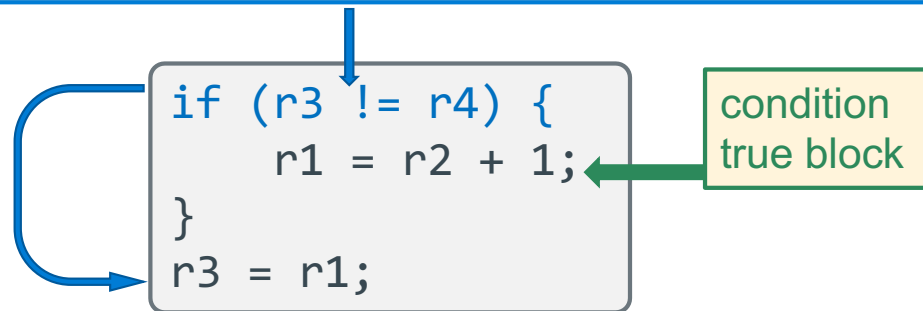
Backward Branch (Infinite loop)

```
.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!

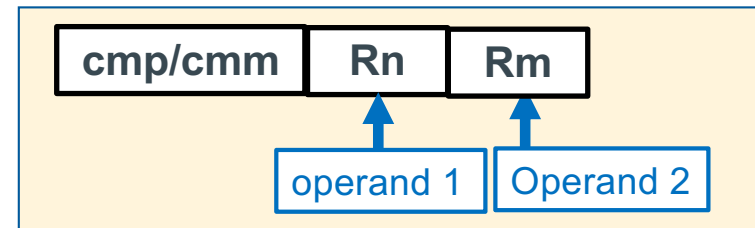
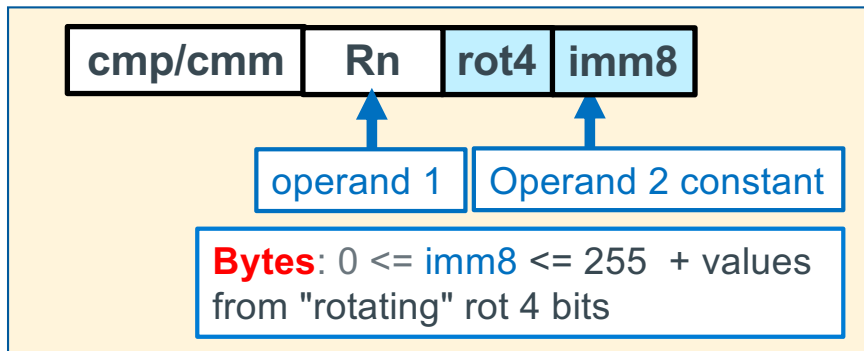
Anatomy of a Sample Branch: Changing the Next Instruction to Execute

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)
- 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

Conditional Branch: Changing the Next Instruction to Execute

cond	b	imm24
------	---	-------

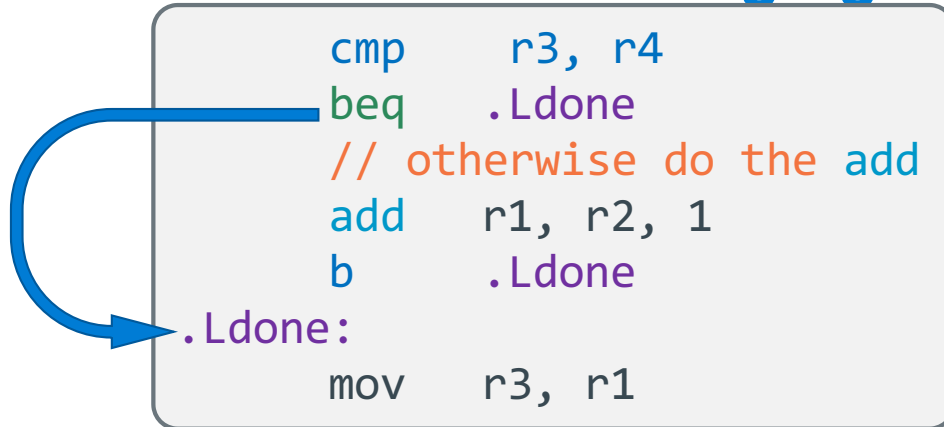
Branch instruction

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)	

Conditional Branch: Changing the Next Instruction to Execute



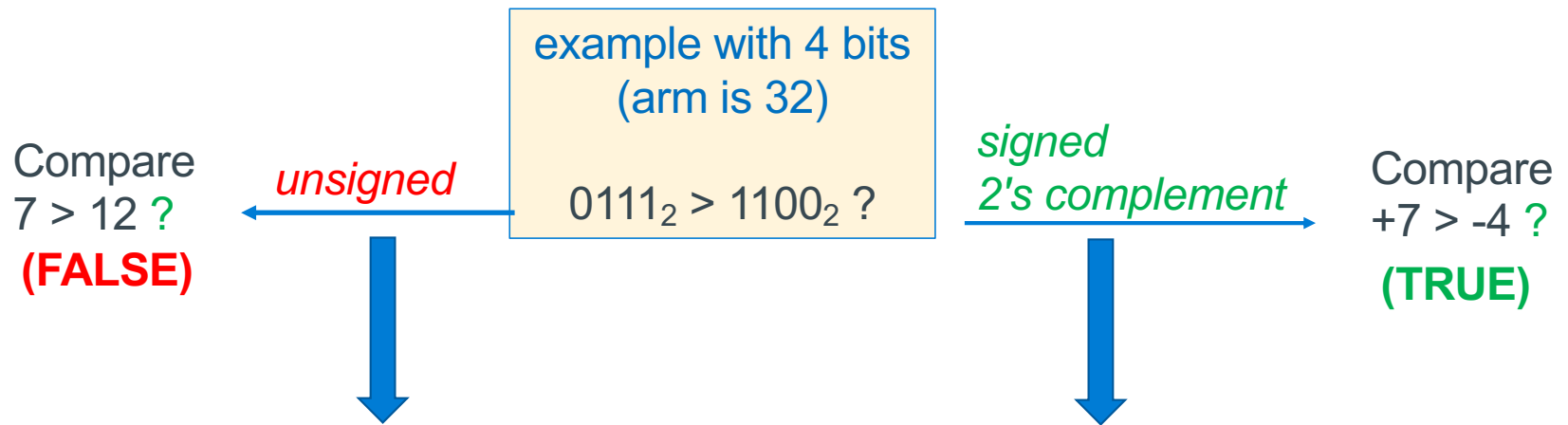
```
if (r3 != r4)
    r1 = r2 + 1;
r3 = r1;
```

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

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>)	

Review Anatomy of a Conditional Branch: If statement

Branch condition
Test (branch guard)

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

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 */  
}
```

condition
true block

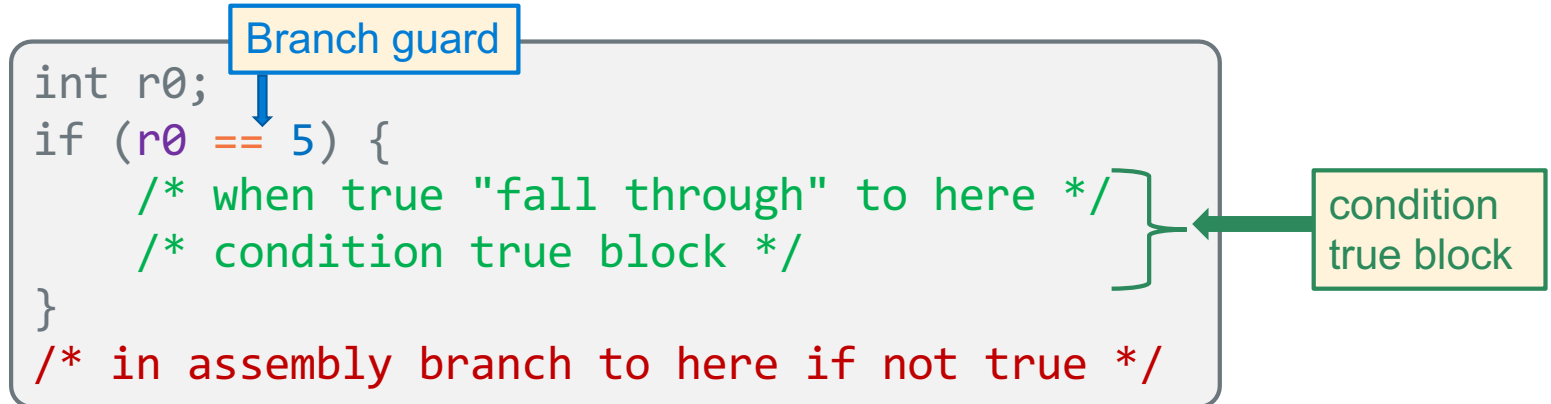
condition
false block

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

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



- In ARM32, you either **fall through** (execute the *next instruction in sequence*) or **branch 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

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 */
}
```

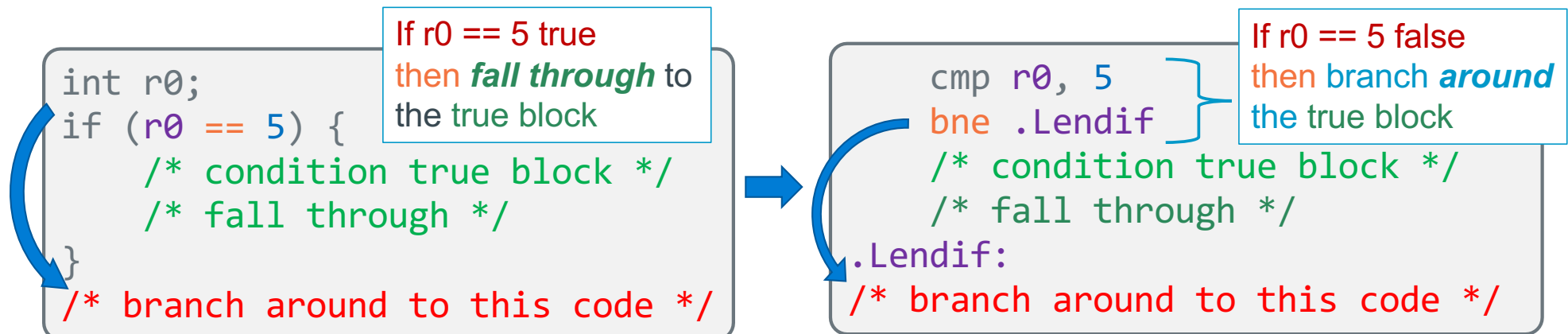
```
cmp r0, 5
inverse .Lelse
    // condition true block
.Lendif:
```

Program Flow: Simple If statement, No Else

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

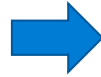
Use a **conditional test** that specifies the **inverse** of the condition used in C

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



If statement examples – Branch Around the True block!

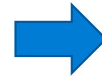
```
int r0;  
if (r0 == 5) {  
    r1 = r2++ + r3;  
}  
r2 = r3;
```



```
cmp    r0, 5  
bne    .Lendif  
add    r1, r2, r3  
add    r2, r2, 1  
.Lendif:  
mov    r3, r2
```

If r0 == 5 false
then branch
around the
true block

```
int r0;  
if (r0 <= 5) {  
    r1 = r2++;  
}  
r2 = r3;
```



```
cmp    r0, 5  
bgt    .Lendif  
mov    r1, r2  
add    r2, r2, 1  
.Lendif:  
mov    r3, r2
```

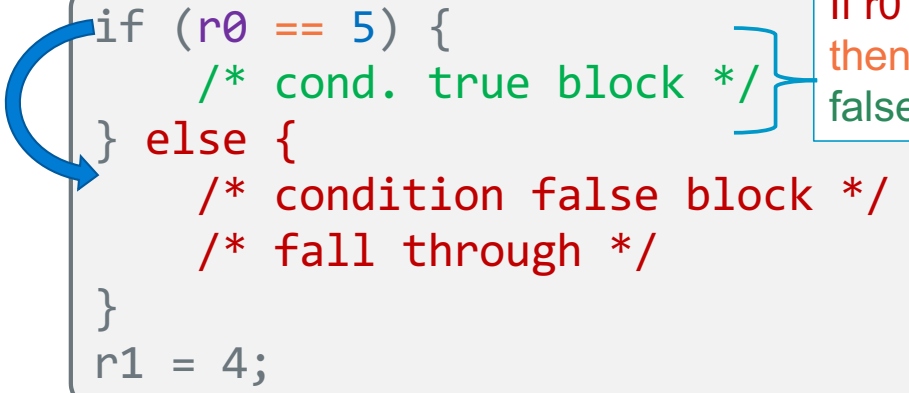
```
unsigned int r0, r1;  
if (r0 > r1) {  
    r1 = r0;  
}  
r2 = r3;
```



```
cmp    r0, r1  
bls    .Lendif  
mov    r1, r0  
.Lendif:  
mov    r3, r2
```

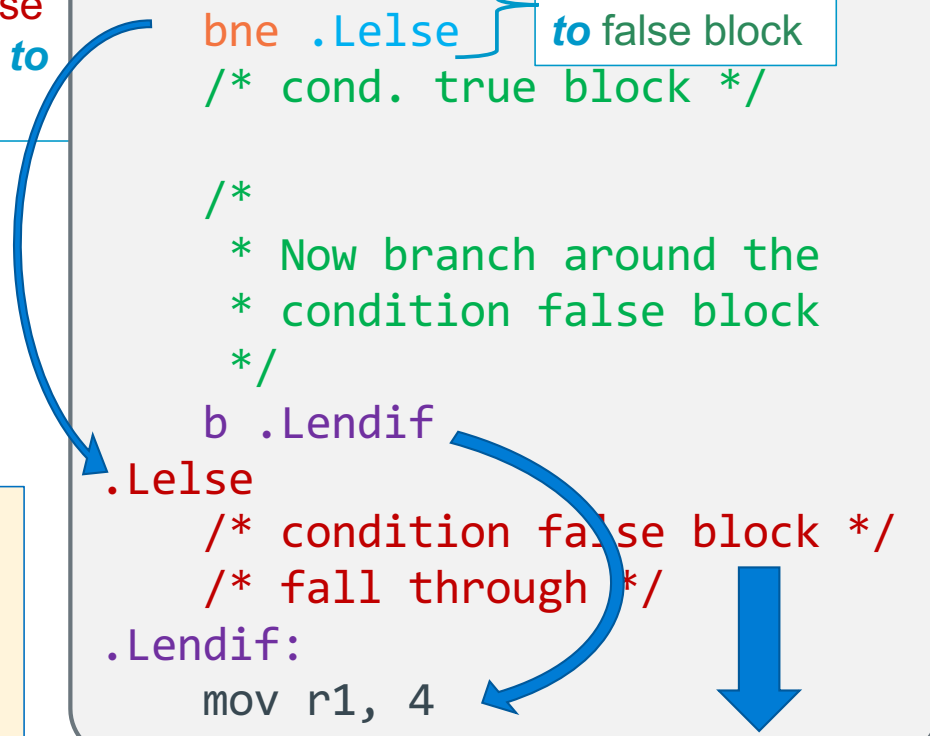
Program Flow: If with an Else

```
if (r0 == 5) {  
    /* cond. true block */  
}  
else {  
    /* condition false block */  
    /* fall through */  
}  
r1 = 4;
```



If r0 == 5 false
then branch to
false block

```
cmp r0, 5  
bne .Lelse  
/* cond. true block */
```




If r0 == 5 false
then branch
to false block

```
/*  
 * Now branch around the  
 * condition false block  
 */
```

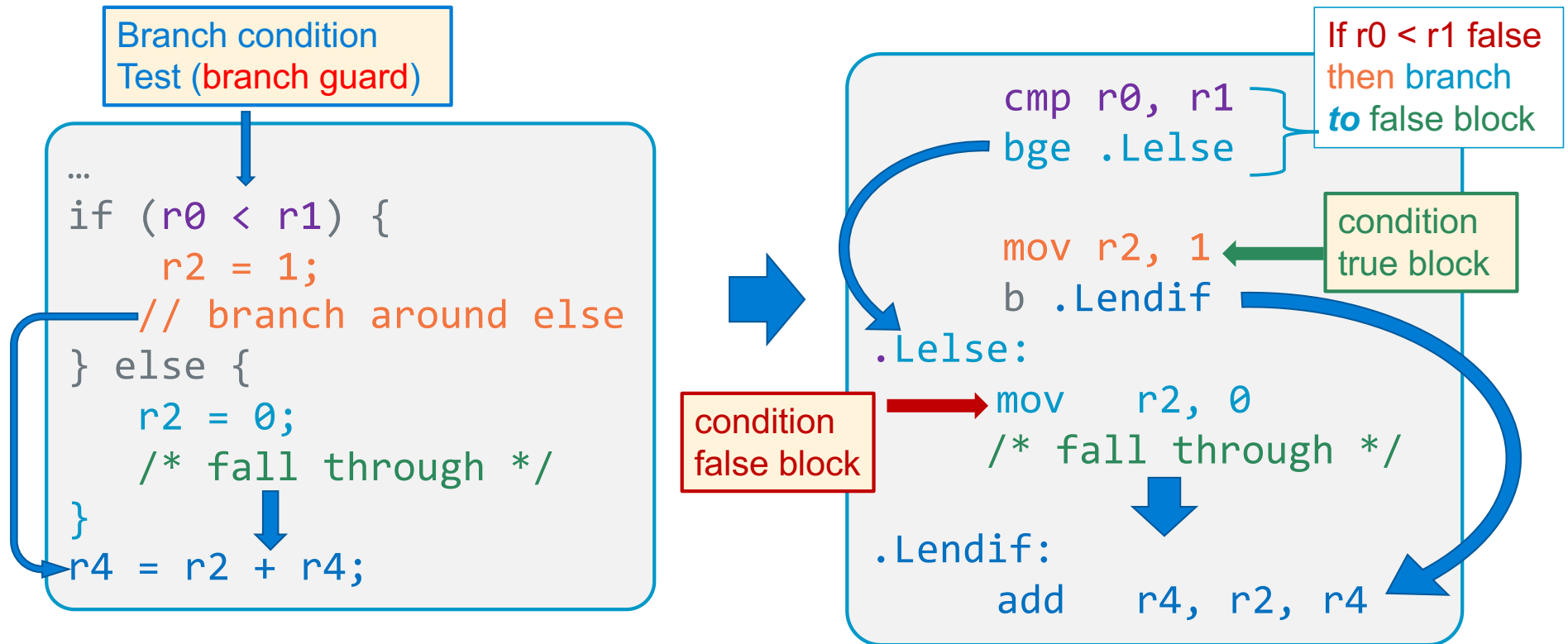
b .Lendif

```
.Lelse  
/* condition false block */  
/* fall through */  
.Lendif:  
mov r1, 4
```

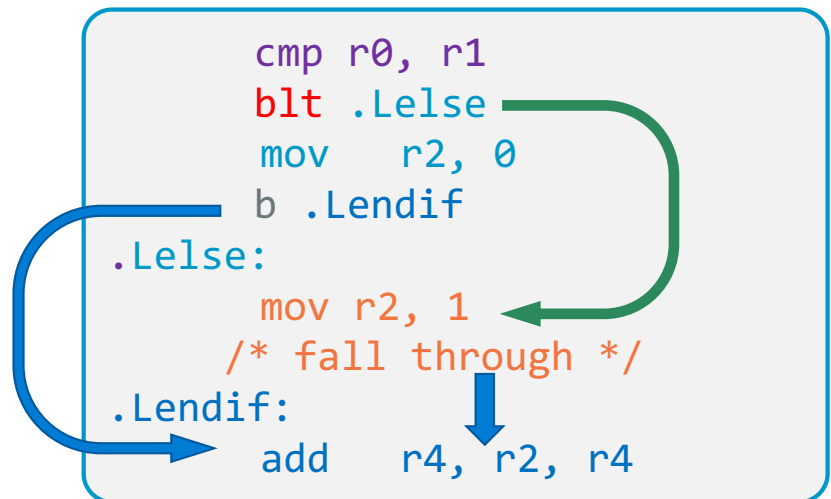
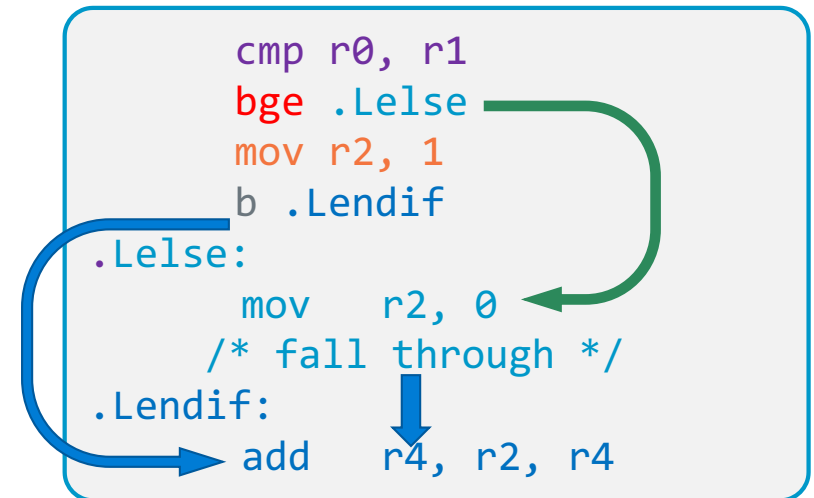
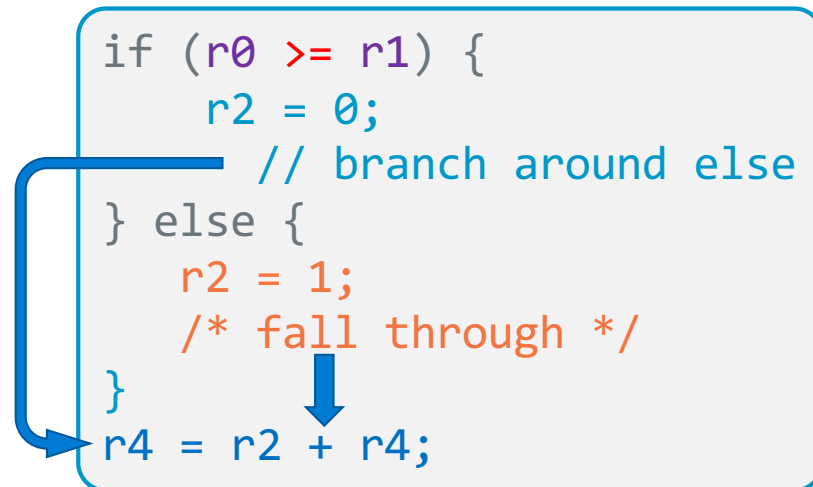
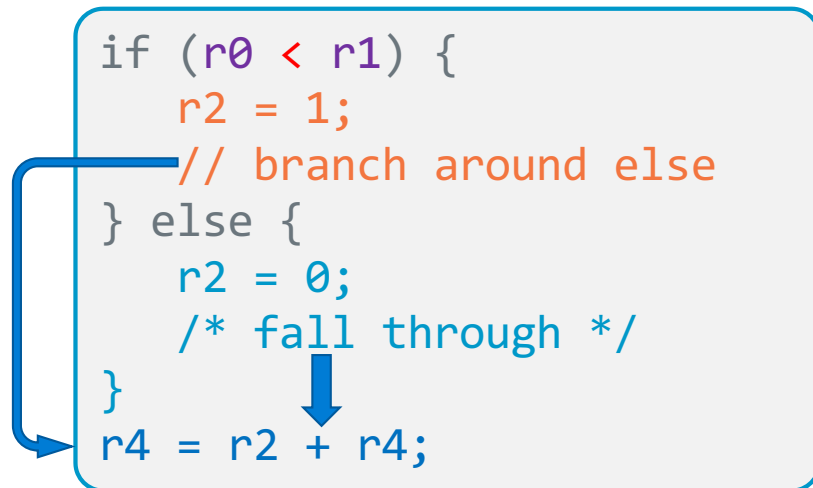


1. Make the adjustment to the conditional test to **branch to** the false block
2. When you finish the true block, you do an **unconditional branch around** the false block
3. The **false block falls through** to the following instructions

If with an Else Examples



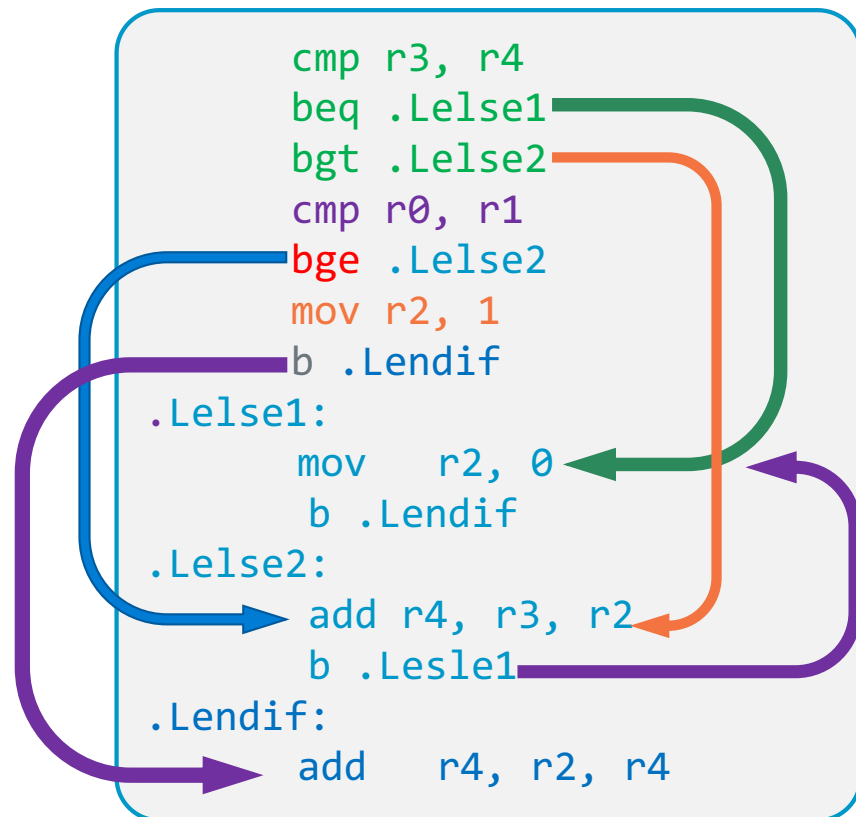
If with an Else Block order: All These Are Equivalent



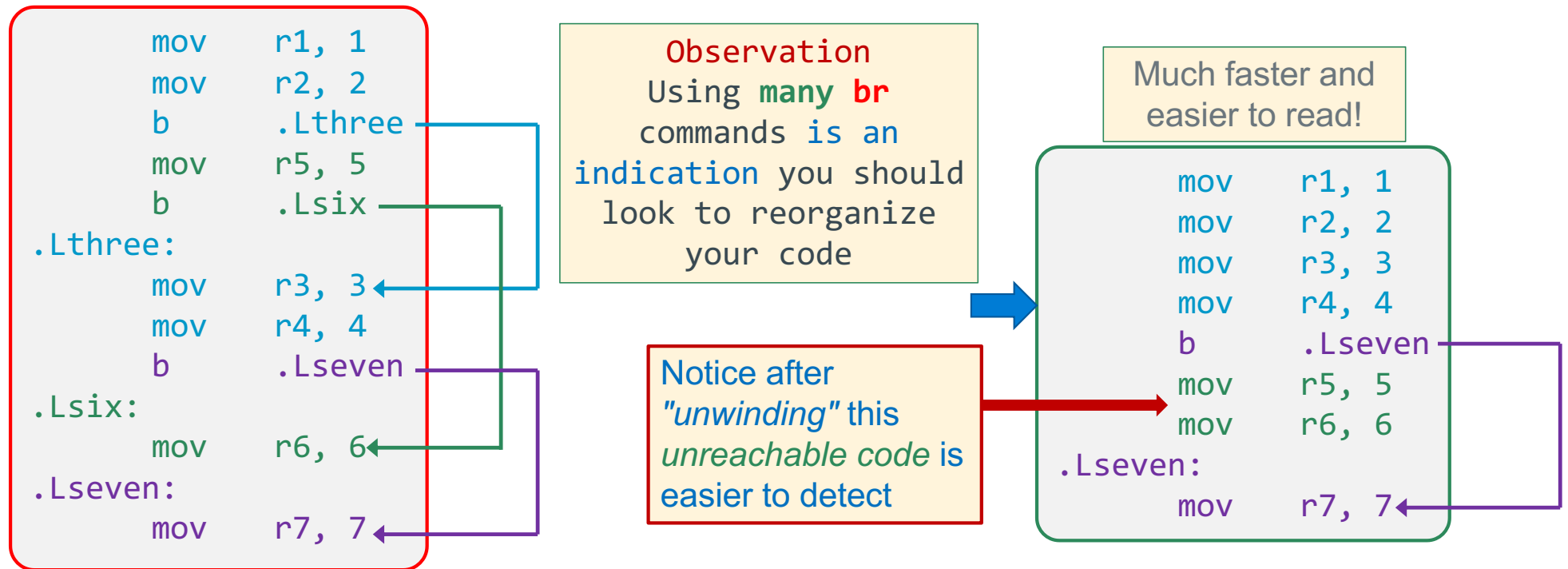
Bad Style: Branching Upwards (Not a loop)

Do not Branch "Upwards" unless it is part of a loop (later slides)

- If you cannot easily write the equivalent C code for your assembly code, you may have code that is harder to read than it should be
- **Action:** adjust your assembly code to have a similar structure as an equivalent version written in C



Branching What not to do: Spaghetti Code



Eliminate unnecessary branches and labels: use Fall Throughs

```
    mov    r2, 0
    b      .Lnext // not needed, slows code
.Lnext:
    add    r1, r2, r3
```

Use fall-through!
do not branch to a Label
on the **next statement!**



```
    mov    r2, 0
    add    r1, r2, r3
```

Branching: Use Fall through!

Some call this "goto like" structure

- Do not use unnecessary branches (sometimes called a “goto” like structure) when a “fall through” works
- You can see this by structures that have a **conditional branch around an unconditional branch that immediately follows it**

Do not do the following:

```
cmp r0, 0
```

```
beq .Lthen
```

```
b .Lendif
```

Not good!

Two adjacent branches

```
.Lthen:
```

```
add r1, r1, 1
```

```
.Lendif:
```

```
add r1, r1, 2
```

Do the following:

```
cmp r0, 0
```

```
bne .Lendif // fall through
```

```
add r1, r1, 1
```

```
.Lendif:
```

```
add r1, r1, 2
```

Switch Statement

Approach 1 – Branch Block

```
switch (r0) {  
  case 1:  
    // block 1  
    break;  
  case 2:  
    // block 2  
    break;  
  default:  
    // default 3  
    break;  
}
```

```
cmp r0, 1  
beq .Lblk1  
cmp r0, 2  
beq .Lblk2
```

Branch
block

```
// fall through  
// default 3  
b .Lendsw // break
```

.Lblk1

```
// block 1  
b .Lendsw // break
```

.Lblk2:

```
// block 2  
// fall through  
// NO b .Lendsw
```

.Lendsw:

Approach 2 – if else equiv.

```
cmp r0, 1  
bne .Lblk2
```

```
// block 1
```

```
b .Lendsw // break
```

.Lblk2:

```
cmp r0, 2  
bne .Ldefault
```

```
// block 2
```

```
b .Lendsw // break
```

.Ldefault:

```
// default 3
```

```
// fall through  
// NO b .Lendsw
```

.Lendsw:

Program Flow – Short Circuit or Minimal Evaluation

- In evaluation of conditional guard expressions, C uses what is called **short circuit** or **minimal evaluation**

```
if ((x == 5) || (y > 3)) // if x == 5 then y > 3 is not evaluated
```



- Each expression argument is evaluated in sequence from left to right including any side effects (modified using parenthesis), before (optionally) evaluating the next expression argument

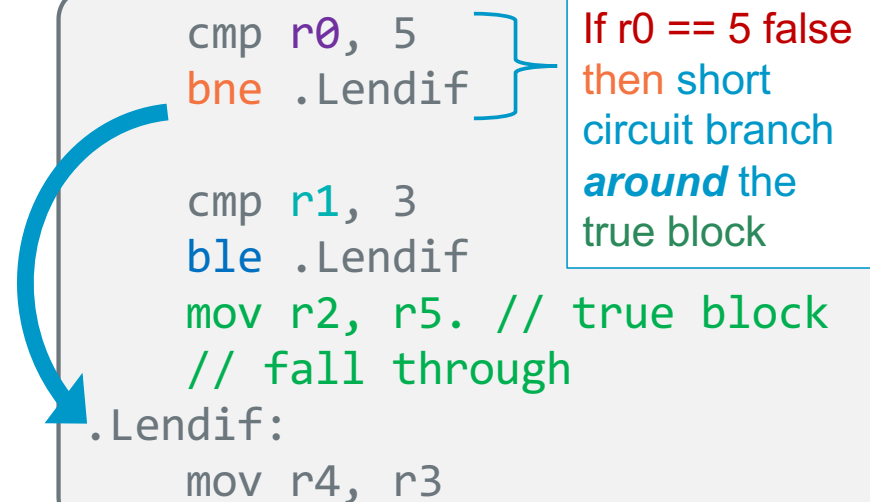
```
if (x || ++x) // true block always executed: ++x!  
    printf("%d\n", x);
```

- If after evaluating an argument, the value of the entire expression can be determined, then the remaining arguments are NOT evaluated (for performance)

```
if ((a != 0) && func(b)) // if a is 0, func(b) is not called  
    // do_something();
```

Program Flow – If statements && compound tests - 1

```
if ((r0 == 5) && (r1 > 3)) {  
    r2 = r5; // true block  
    /* fall through */  
}  
r4 = r3;
```

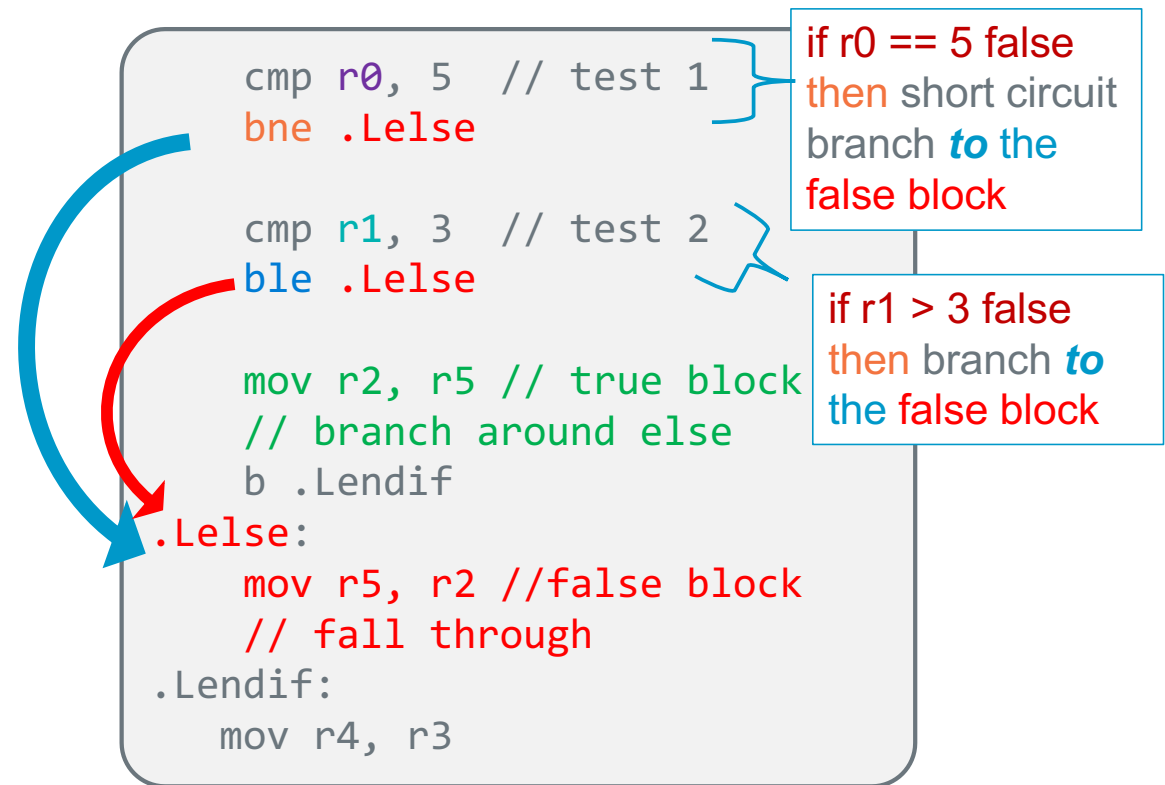


```
cmp r0, 5  
bne .Lendif  
  
cmp r1, 3  
ble .Lendif  
mov r2, r5. // true block  
// fall through  
.Lendif:  
mov r4, r3
```

If $r0 == 5$ false
then short
circuit branch
around the
true block

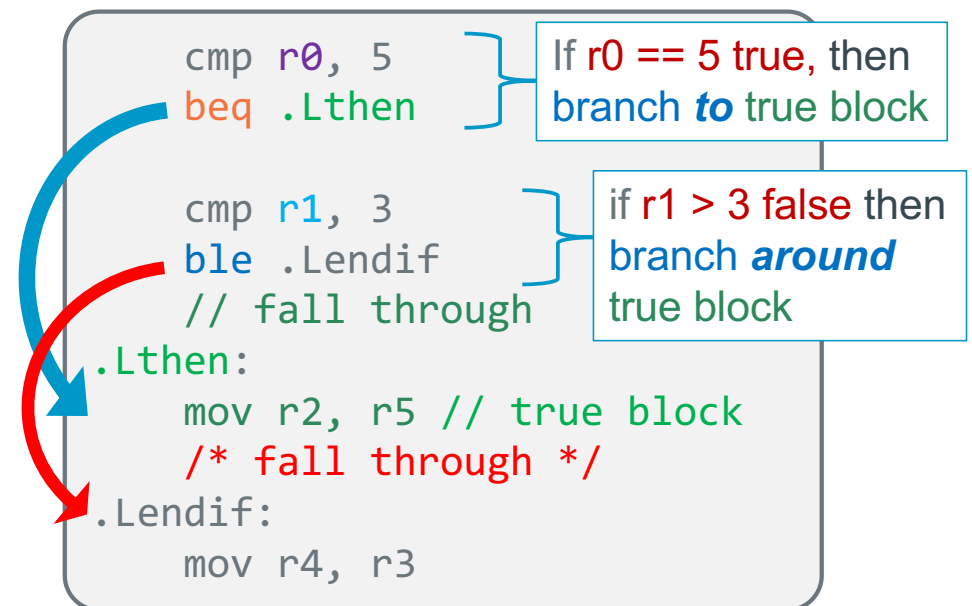
Program Flow – If statements && compound tests - 2

```
if ((r0 == 5) && (r1 > 3))
{
    r2 = r5; // true block
    // branch around else
} else {
    r5 = r2; False block */
    /* fall through */
}
r4 = r3;
```



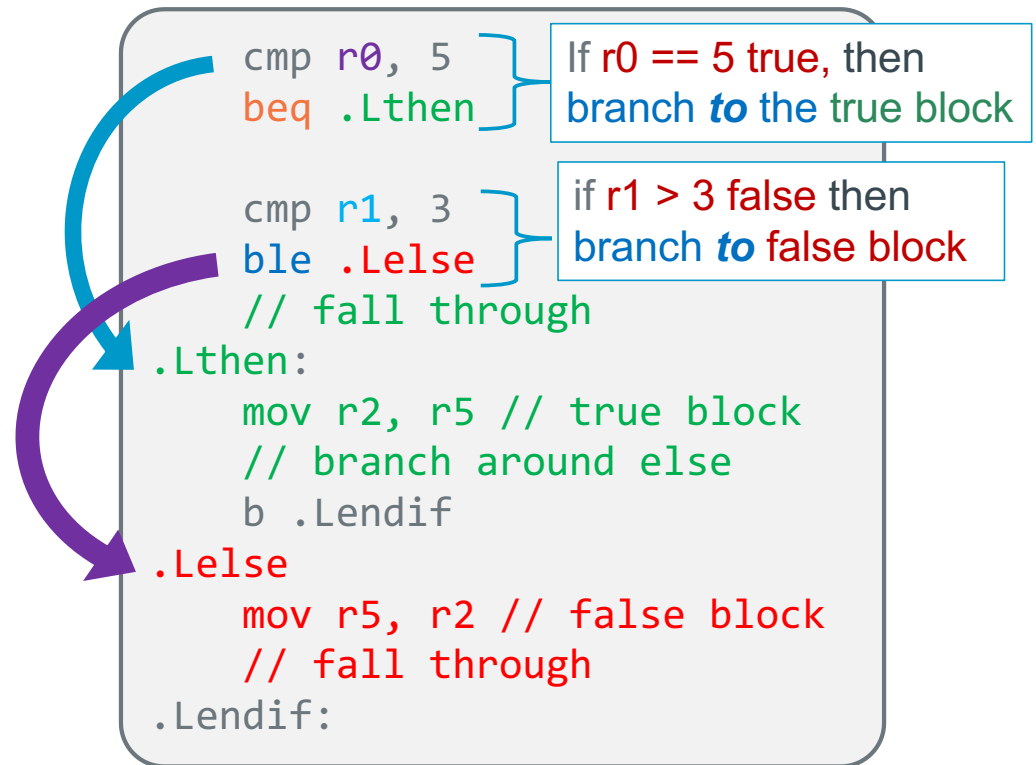
Program Flow – If statements || compound tests - 1

```
if ((r0 == 5) || (r1 > 3)) {  
    r2 = r5; // true block  
    /* fall through */  
}  
r4 = r3;
```



Program Flow – If statements || compound tests - 2

```
if ((r0 == 5) || (r1 > 3)) {  
    r2 = r5; // true block  
    /* branch around else */  
} else {  
    r5 = r2; // false block  
    /* fall through */  
}
```




Program Flow – multiple branches, one cmp

```
if ((r0 > 5) {  
    /* condition block 1 */  
    // branch to endif  
} else if (r0 < 5){  
    /* condition block 2 */  
    // branch to endif  
} else {  
    /* condition block 3 */  
    // fall through to endif  
}  
// endif  
r1 = 11;
```

- There are many other ways to do this

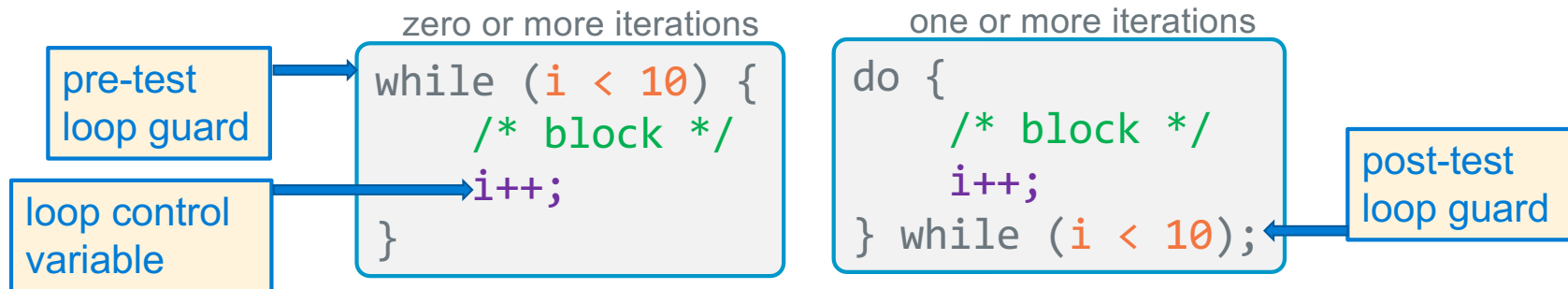
```
cmp r0, 5  
bgt .Lblk1  
blt .Lblk2  
// fall through  
// condition block 3  
b .Lendif  
.Lblk1:  
    // condition block 1  
    b .Lendif  
.Lblk2:  
    // condition block 2  
    b .Lendif  
.Lendif:  
    mov r1, 5
```



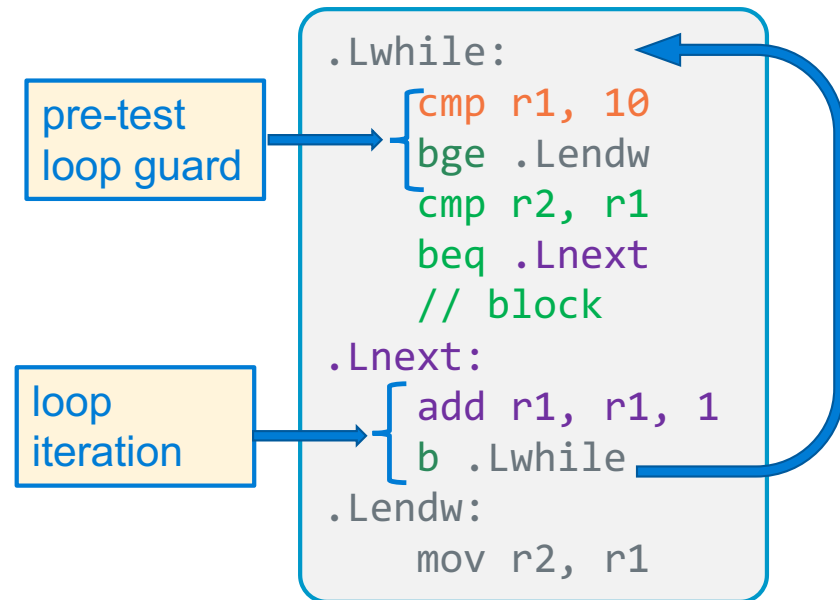
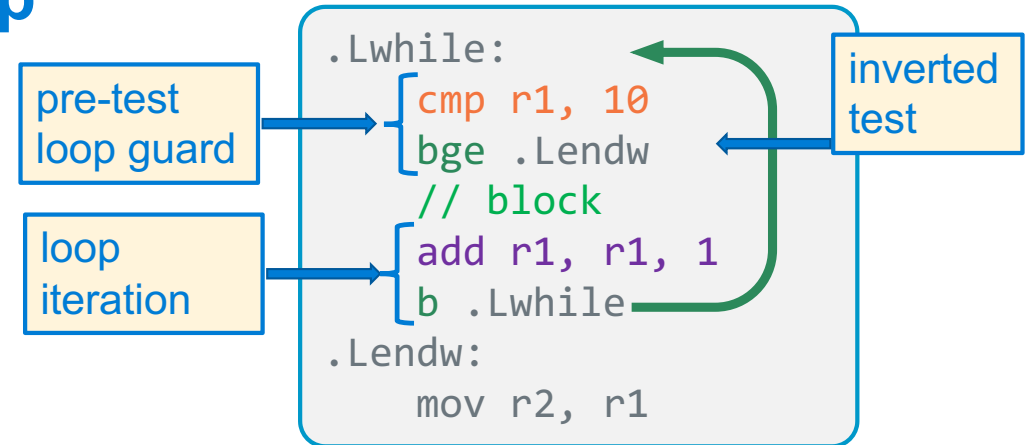
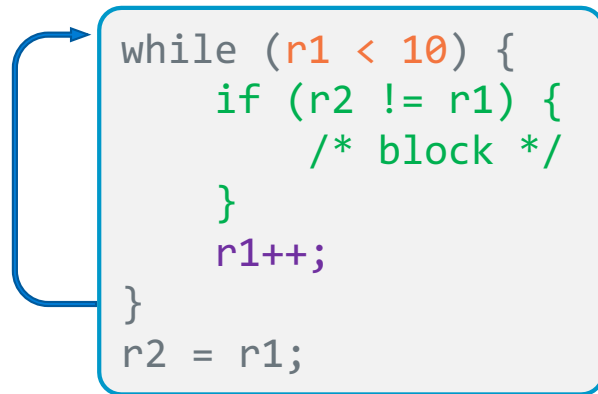
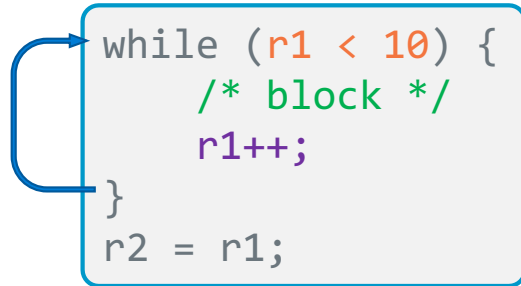
The diagram illustrates the flow of execution. A green arrow points from the `bgt .Lblk1` instruction to the `.Lblk1:` label. A purple arrow points from the `blt .Lblk2` instruction to the `.Lblk2:` label. Both `.Lblk1:` and `.Lblk2:` have a `b .Lendif` instruction, which then leads to the `.Lendif:` label. A yellow box highlights the `bgt .Lblk1` and `blt .Lblk2` instructions with the text "special case: multiple branches from one cmp".

Program Flow – Pre-test and Post-test Loop Guards

- loop guard: code that must evaluate to true before the next iteration of the loop
- If the loop guard test(s) evaluate to true, the *body of the loop* is executed again
- pre-test loop guard is at top of the loop
 - If the test evaluates to true, execution falls through to the loop body
 - if the test evaluates to false, execution branches around the loop body
- post-test loop guard is at the bottom of the loop
 - If the test evaluates to true, execution branches to the top of the loop
 - If the test evaluates to false, execution falls through the instruction following the loop



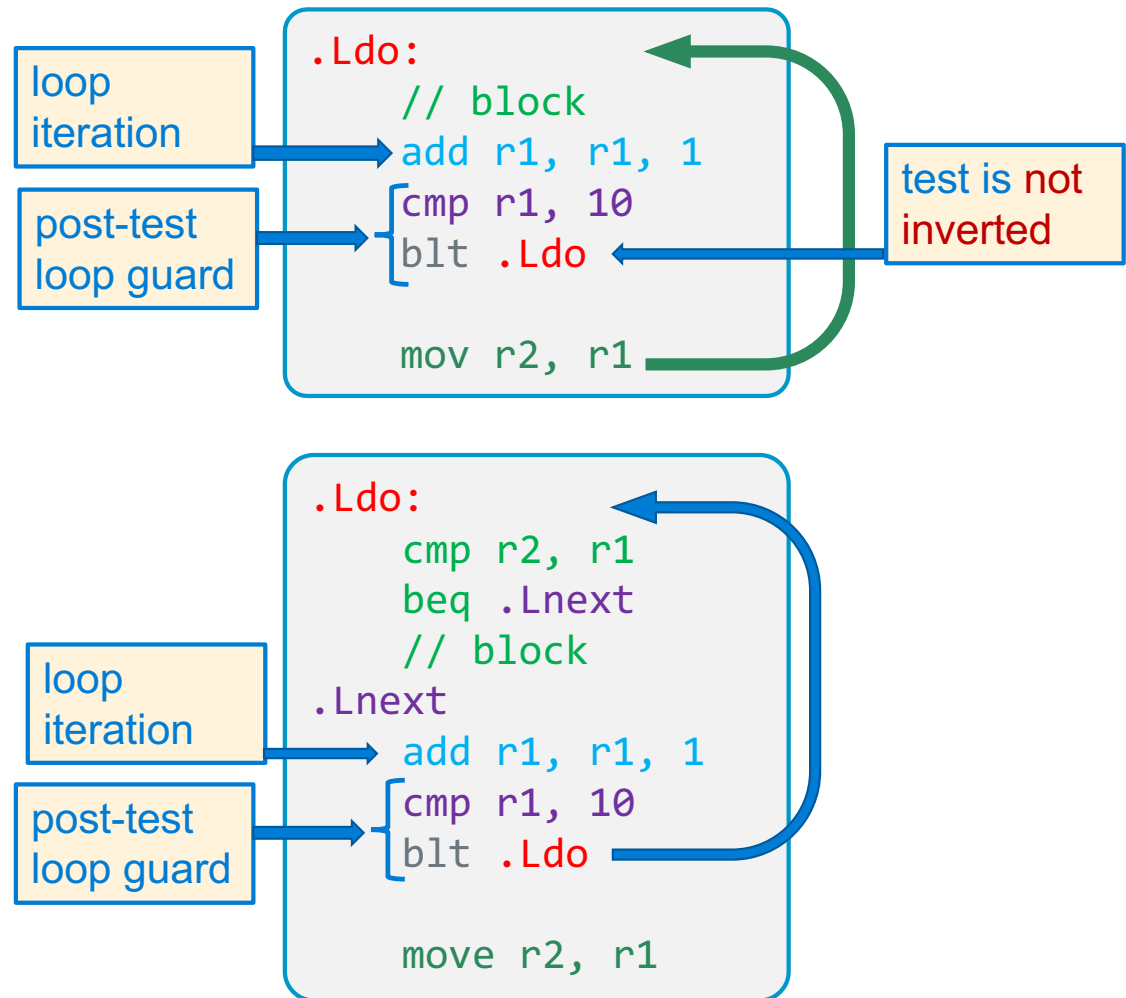
Pre-Test Guards - While Loop



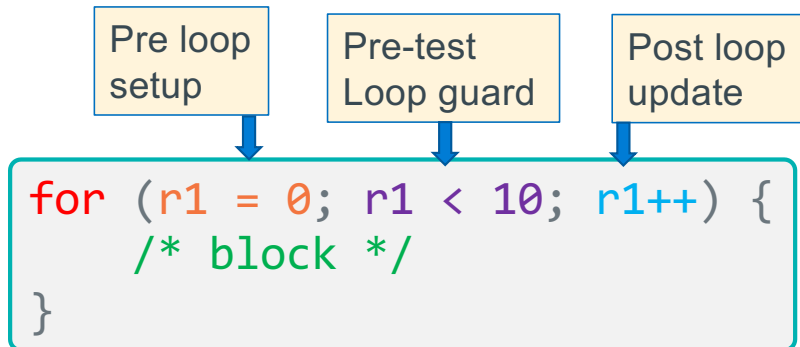
Post-Test Guards – Do While Loop

```
do {  
    /* block */  
    r1++;  
} while (r1 < 10);  
  
r2 = r1;
```

```
do {  
    if (r2 != r1) {  
        /* block */  
    }  
    r1++;  
} while (r1 < 10);  
  
r2 = r1;
```

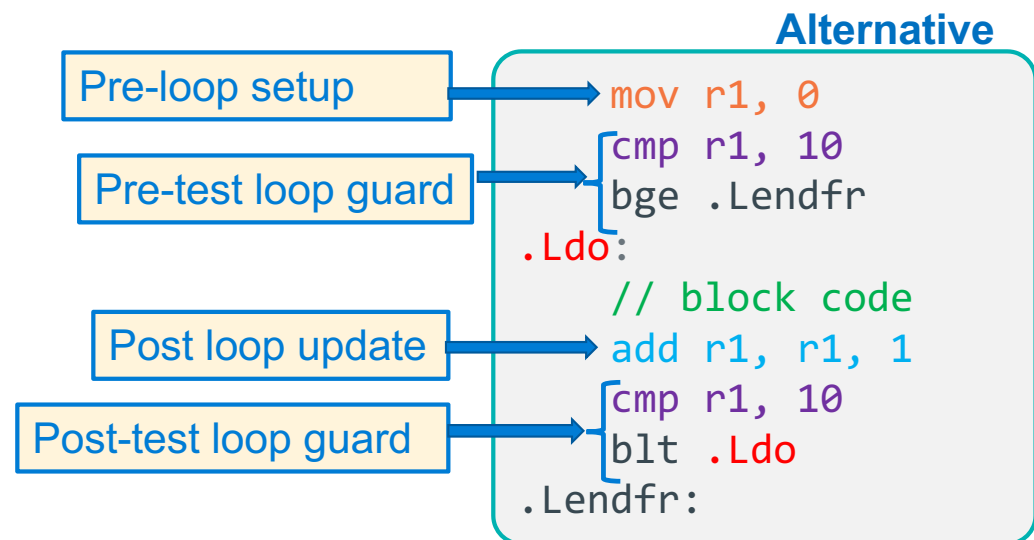
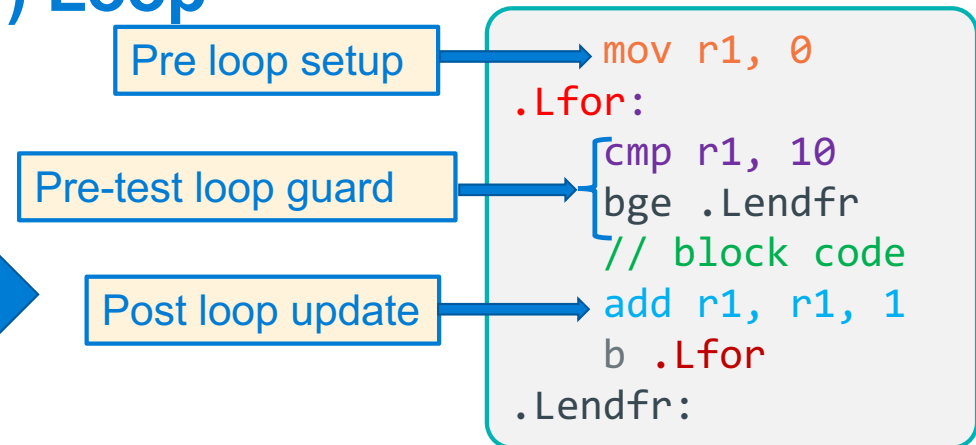


Program Flow – Counting (For) Loop




A **counting loop** has three parts:

1. **Pre-loop** setup
 2. **Pre-test loop guard** conditions
 3. **Post-loop** update
- **Alternative:**
 - **move** Pre-test loop guard **before** the loop
 - **Add** post-test loop guard
 - **converts** to **do while**
 - **removes** an **unconditional branch**



Nested loops


```
for (r3 = 0; r3 < 10; r3++) {  
    r0 = 0;  
  
    do {  
        r0 = r0 + r1++;  
    } while (r1 < 10);  
  
    // fall through  
    r2 = r2 + r1;  
}
```



r5 = r0;

- Nest loop blocks as you would in C or Java
- Do not branch into the middle of a loop, this is hard to read and is prone to errors

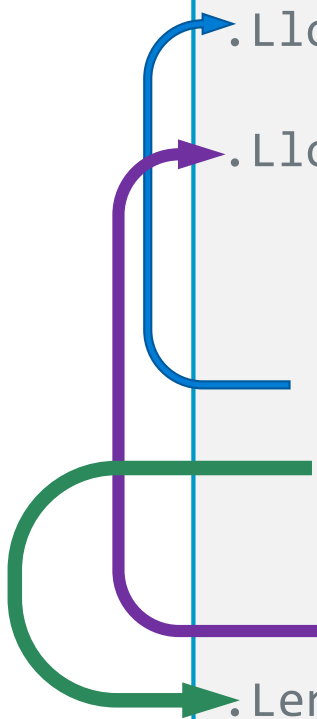
```
mov r3, 0  
.Lfor:  
    cmp r3, 10        // loop guard  
    bge .Lendfor  
  
    mov r0, 0  
  
    .Ldo:  
        add r0, r0, r1  
        add r1, r1, 1  
  
        cmp r1, 10    // loop guard  
        blt .Ldo  
  
        // fall through  
        add r2, r2, r1  
  
        add r3, r3, 1 // loop iteration  
        b .Lfor  
    .Lendfor:  
        mov r5, r0
```



Keeps loops Properly Nested

- It is hard to understand and debug loops when the "branch into each other"
- **Keep loops proper nested**

Do not do the following:



```
.Lloop1:  
    add r1, r1, 1  
.Lloop2:  
    add r2, r2, 1  
    add r2, r1, r3  
    cmp r1, 10  
    blt .Lloop1  
    beq .Lend1  
    add r3, r3, 1  
    cmp r2, 20  
    ble .Lloop2  
.Lend1:
```

The diagram illustrates a non-nested loop structure. A blue arrow shows a jump from the end of the first loop back to its start. A purple arrow shows a jump from the end of the second loop back to the start of the first loop. A green arrow shows a jump from the end of the second loop to the end of the entire loop structure. This structure is problematic because the loops are not properly nested, making them difficult to understand and debug.

Creating Segments, Definitions In Assembly Source

- The following assembler directives indicate the **start** of a **memory segment specification**
 - **Remains in effect** until the next segment directive is seen

```
.bss
    // start uninitialized static segment variables definitions
    // does not consume any space in the executable file
.data
    // start initialized static segment variables definitions
.section .rodata
    // start read-only data segment variables definitions
.text
    // start read-only text segment (code)
```

- Define a **literal**, **static variable** or **global** variable in a segment

```
Label:    .size_directive expression, ... expression
```

- **Label**: this is the **variables name**
- **Size_Directive** tells the **assembler** *how much space to **allocate*** for that **variable**
- Each **optional expression** specifies the contents of one memory location of **.size_directive**
 - **expression** can be in **decimal**, **hex** (0x...), **octal** (0...), **binary** (0b...), **ASCII** (' '), **string** " "

```

// File Header
.arch armv6           // armv6 architecture instructions
.arm                 // arm 32-bit instruction set
.fpu vfp             // floating point co-processor
.syntax unified       // modern syntax

// BSS Segment (only when you have initialized globals)
.bss
// Data Segment (only when you have uninitialized globals)
.data
// Read-Only Data (only when you have literals)
.section .rodata
// Text Segment - your code
.text

// Function Header
.type main, %function // define main to be a function
.global main           // export function name
main:
// function prologue           // stack frame setup
                                // your code for this function here
// function epilogue           //stack frame teardown

// function footer
.size main, (. - main)

// File Footer
.section .note.GNU-stack,"",%progbits // stack/data non-exec
.end

```

Assembly Source File

- assembly programs end in **.S**
 - **example:** test.S
- Always use gcc to assemble
 - `_start()` and C runtime
- File has a complete program
 - `gcc file.S`
- File has a partial program
 - `gcc -c file.S`
- Link files together
 - `gcc file.o cprog.o`

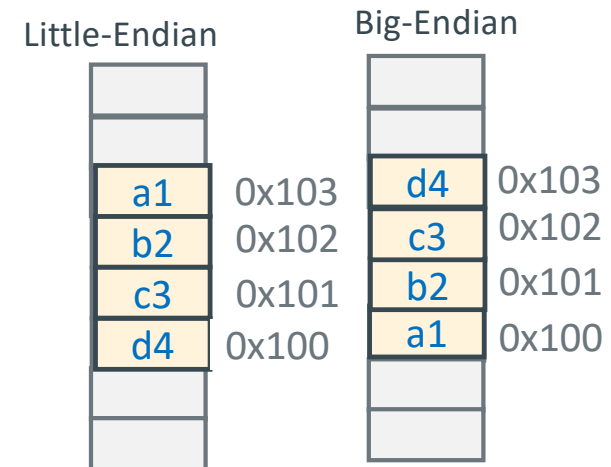
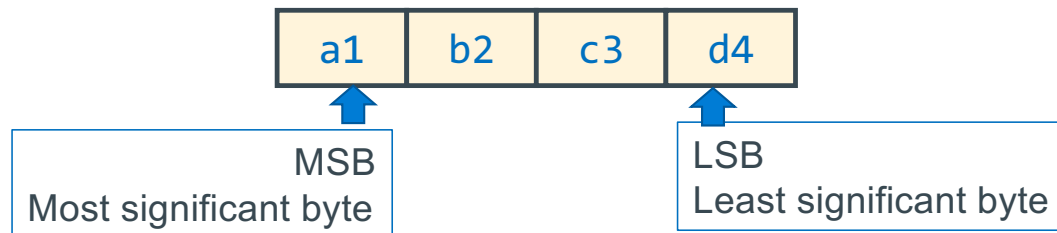
Memory Segment Data Alignment

- **Word** is the **number of bytes** necessary to store an **address** (32-bits on Pi-cluster) – **hardware defined**
- The **address** of **any sized** unit of memory is always the **address** of the **first byte**
- Hardware often requires Variables to be *"aligned"* to specific starting addresses based on type
- char (1 byte)
 - can start at any address
- short (2 bytes) can start only at addresses ending
 - b..00 or b..10 (.align 1) // last **bit** must be 0
- int (4 bytes) can start only at address ending in
 - 0b..00 (.align 2) // last **two bits** must be 0

32-bit units (4 bytes)	16-bit units (2 Bytes)	8-bit units (1 Byte)	Addr. (binary)
	Start at b..10		b..10011
Start At b..00	Start at b..00		b..10010
	Start at b..10		b..10001
	Start at b..00		b..10000
Start at b..00	Start at b..10		b..01111
	Start at b..00		b..01110
	Start at b..10		b..01101
	Start at b..00		b..01100
	Start at b..10		b..01011
Start at b..00	Start at b..00		b..01010
	Start at b..10		b..01001
	Start at b..00		b..01000
	Start at b..10		b..00111
Start at b..00	Start at b..00		b..00110
	Start at b..10		b..00101
	Start at b..00		b..00100

Byte Ordering of Numbers In Memory: Endianness

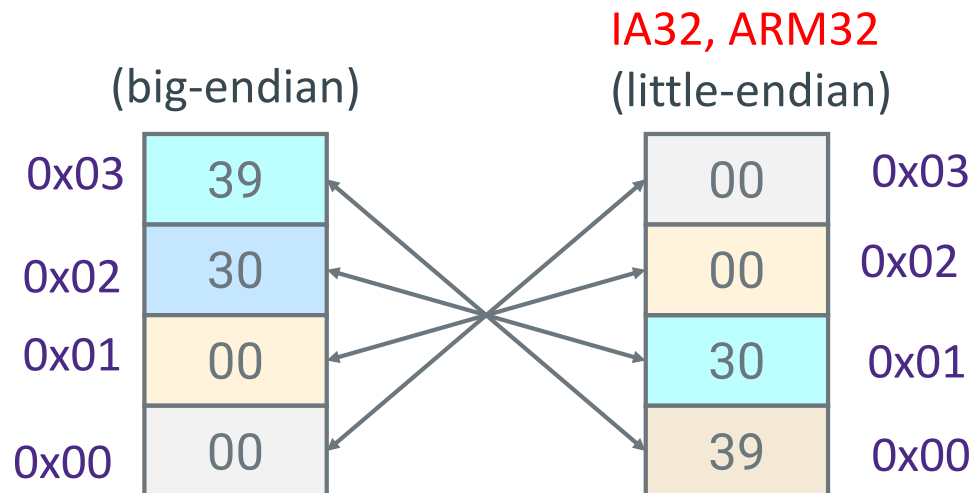
- Two different ways to place multi-byte integers in a **byte addressable** memory
- **Big-endian**: **Most** Significant Byte (“**big end**”) starts at the **lowest (starting)** address
- **Little-endian**: **Least** Significant Byte (“**little end**”) starts at the **lowest (starting)** address
- Example: 32-bit integer with 4-byte data



Byte Ordering Example

Decimal:	12345
Binary:	0011 0000 0011 1001
Hex:	3 0 3 9

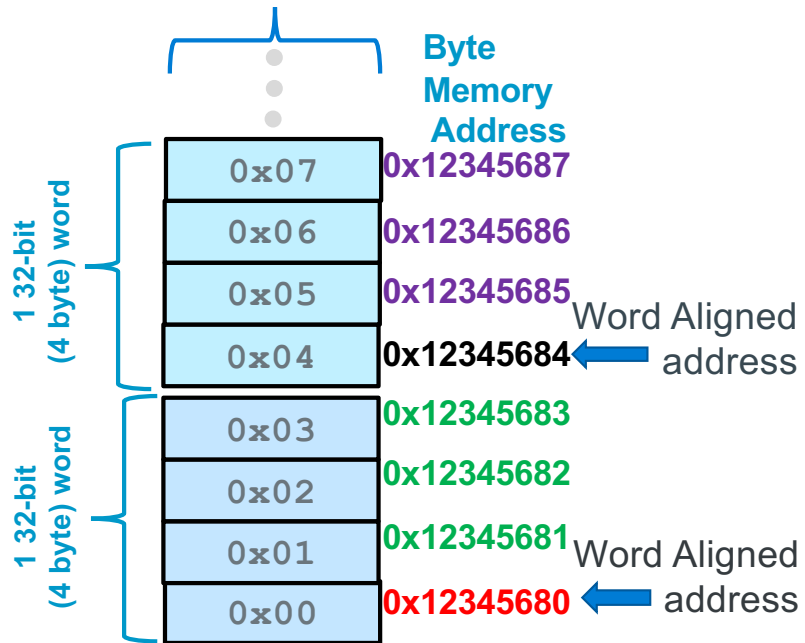
```
int x = 12345;  
// or x = 0x3039;
```



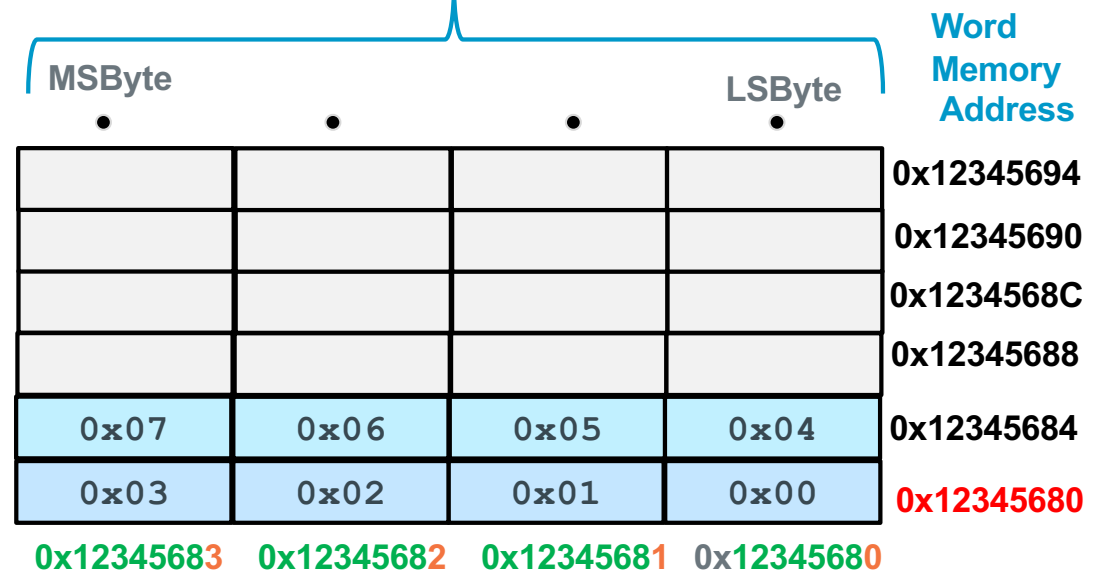
Byte Addressable Memory Shown as 32-bit words

1 byte Memory Content

One byte per row



Contents of Memory
One 32-bit (4 byte) word per row



Byte address

Observation
32-bit aligned addresses
rightmost 2 bits of the address are always 0

Defining Static Variables: Allocation and Initialization

Variable SIZE	Directive	Align	C static variable Definition	Assembler static variable Definition
8-bit char (1 byte)	.byte		char chx = 'A' char string[] = { 'A', 'B', 'C', 0 };	chx: .byte 'A' string: .byte 'A', 'B', 0x42, 0
16-bit int (2 bytes)	.hword .short	1	short length = 0x55aa;	length: .hword 0x55aa
32-bit int (4 bytes)	.word .long	2	int dist = 5; int *distptr = &dist; int array[] = { 12, ~0x1, 0xCD, -1 };	dist: .word 5 distptr: .word dist array: .word 12, ~0x1, 0xCD, -3
strings '\0' term	.string		char class[] = "cse30";	class: .string "cse30"

```
int num;
int *ptr = &num;
char msg[] = "123";
char *lit = "456";
```



```
.bss
num: .word 0
.data
ptr: .word num
msg: .string "123"
lit: .word .Lmsg
.section .rodata
.Lmsg: .string "456"
```

Defining Array Static Variables

Label: .size_directive expression, ... expression

```
In C:      int int_buf[100];
           int array[] = {1, 2, 3, 4, 5};
           char buffer[100];

.bss
int_buf:   .space 400    // convert 100 to 400 bytes
char_buf:  .space 100

.data
array:     .word 1, 2, 3, 4, 5
one_buf:   .space 100, 1 // 100 bytes each byte filled with 1
```

.space size, fill

- Allocates **size** bytes, each of which contain the value **fill**
- Both **size** and **fill** are absolute expressions
- If the comma and **fill** are **omitted**, **fill** is assumed to be **zero**
- **.bss section**: Must be used **without a specified fill**

Static Variable Alignment: Using .align

Accessing **address aligned** memory based on data type has the best performance

integer

4 bytes

short

2 bytes

char

1

SIZE	Directive	Address ends in	Align Directive
8-bit char -1 byte	.byte	0b..0 or 0b..1	
16-bit int -2 bytes	.hword .short	0b..0	.align 1
32-bit int -4 bytes	.word .long	0b..00	.align 2

.align n before variable definition to specify memory alignment requirements

- Tells the assembler the **next line that allocates memory** must **start** at the next higher memory address where the **lower n address bits** are zero
- At the **first use of any Segment directive**, alignment **starts at an 8-byte aligned address** (for doubles)
- **Easy approach:** Allocate from largest size to smaller size

4 bytes	2 Bytes	1 Byte	Addr. (hex)
	Addr = 0x0E		0x0F
			0x0E
Addr = 0x0C	Addr = 0x0C		0x0D
			0x0C
	Addr = 0x0A		0x0B
Addr = 0x08	Addr = 0x08		0x0A
			0x09
			0x08
	Addr = 0x06		0x07
Addr = 0x04	Addr = 0x04		0x06
			0x05
			0x04
	Addr = 0x02		0x03
Addr = 0x00	Addr = 0x00		0x02
			0x01
			0x00

X

Data Segments Alignment

```
.data
ch:      .byte 'A','B','C','D','E'
str:     .string "HIT"
ary:     .hword 0, 1
a:       .byte 'A'
b:       .byte 'B'
xx:      .word 2
```

- pass args to gas with `-Wa,-<gas_args>`
- Use: `%gcc -c -Wa,-ahlns all.s`

```
.data
xx:      .word 2
ch:      .byte 'A','B','C','D','E'
         .align 2
str:     .string "HI"
         .align 1
ary:     .hword 0, 1
a:       .byte 'A'
b:       .byte 'B'
```

```
% gcc -c -Wa,-ahlns all.s
1          .data
2 0000 41424344 ch:      .byte 'A','B','C','D','E'
2          45
3 0005 48495400 str:    .string "HIT"
4 0009 00000100 ary:    .hword 0, 1
5 000d 41          a:    .byte 'A'
6 000e 42          b:    .byte 'B'
8 000f 02000000 xx:    .word 2
```

address contents

```
gcc -c -Wa,-ahlns all.s
1          .data
2 0000 02000000 xx:    .word 2
3 0004 41424344 ch:    .byte 'A','B','C','D','E'
3          45
4 0009 000000      .align 2
5 000c 484900      str:  .string "HI"
6 000f 00          .align 1
7 0010 00000100 ary:  .hword 0, 1
8 0014 41          a:    .byte 'A'
9 0015 42          b:    .byte 'B'
```

Load/Store: Register Base Addressing

ldr r0, [r1]

Copies a 32-bit word from the memory location whose address is contained in r1 (r1 is a pointer) into register r0

32-bit memory



register r0

register r1 (address)



r1 is being used as a pointer to a location in memory

ldr requires the use of a pointer operand

str r0, [r1]

Copies all 32 bits of the value held in register r0 to the 32-bit memory location contained in register r1 (r1 pointer)

register r0



32-bit memory

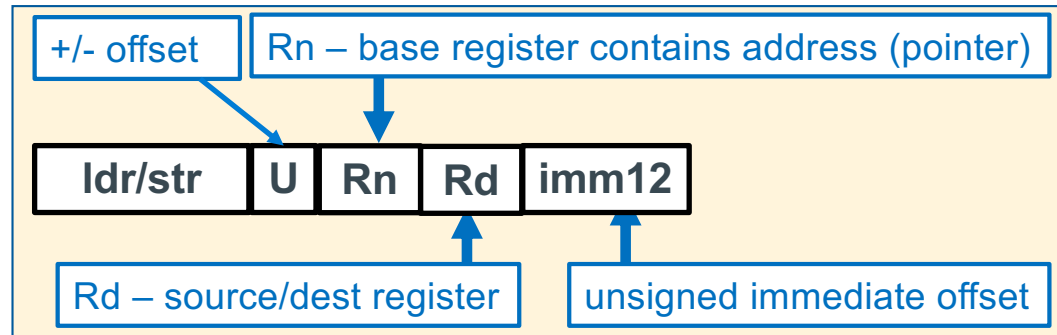
r1 is being used as a pointer to a location in memory

str requires the use of a pointer operand

register r1 (address)



LDR/STR – Base Register + Immediate Offset Addressing



- **Register Base Addressing:**

- **Pointer Address:** Rn; **source/destination data:** Rd
- **Unsigned pointer address** is stored in the **base register**

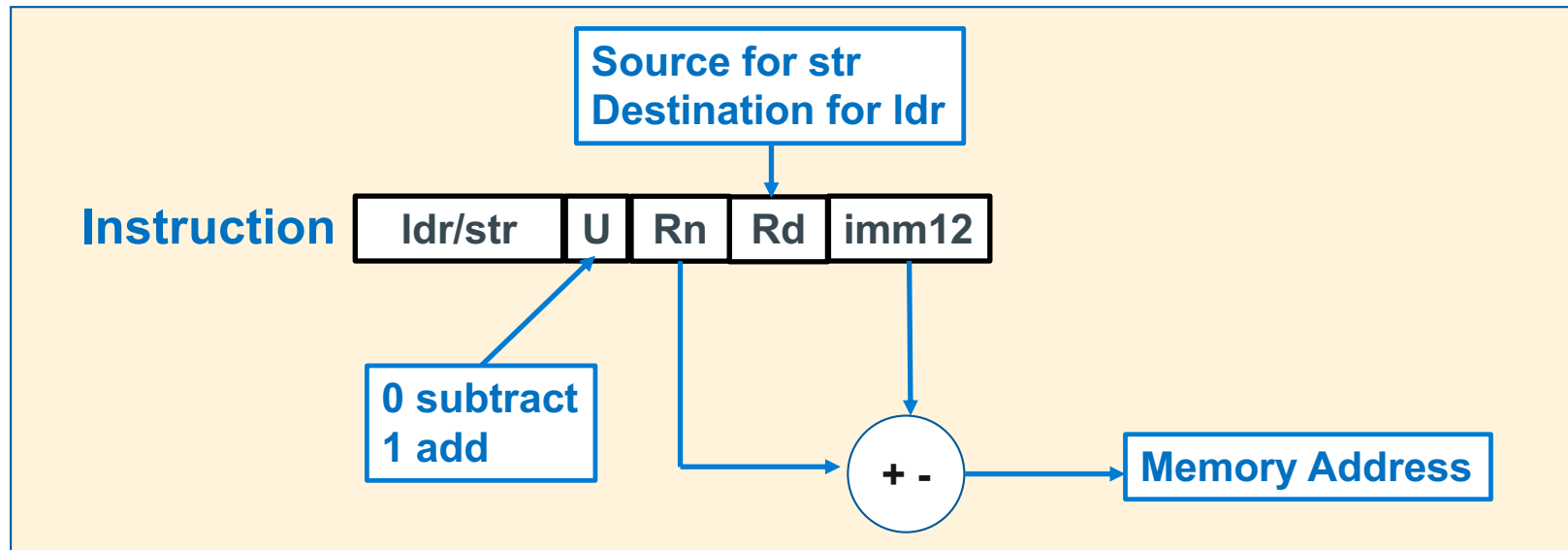
- **Register Base + immediate offset Addressing:**

- **Pointer Address** = register content + immediate offset
- **Unsigned offset integer immediate value (bytes)** is added or subtracted (**U bit above says to add or subtract**) from the **pointer address** in the **base register**

```
ldr/str  Rd,  [Rn, +/- imm12] // base register pointer + offset  imm12 in bytes
                                     -4095 <= imm12 <= 4095 (bytes)

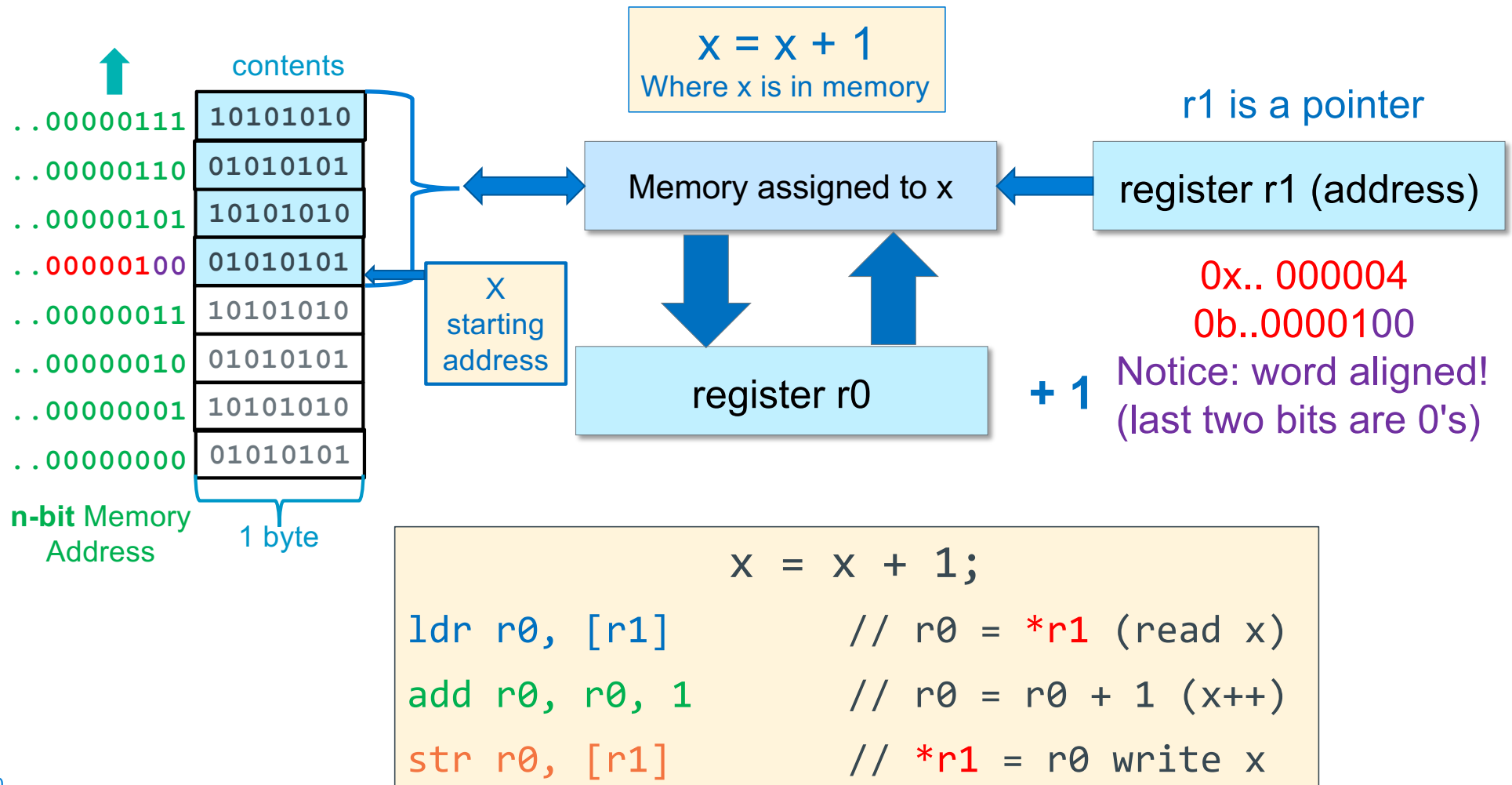
ldr/str  Rd,  [Rn]             // base register pointer + 0 offset (imm12 is 0)
```

ldr/str Register Base and Register + Immediate Offset Addressing



Syntax	Address	Examples
<code>ldr/str Rd, [Rn +/- constant]</code> constant is in bytes	<code>Rn + or - constant</code> same \longrightarrow	<code>ldr r0, [r5,100]</code> <code>str r1, [r5, 0]</code> <code>str r1, [r5]</code>

Example Base Register Addressing Load – Modify – Store



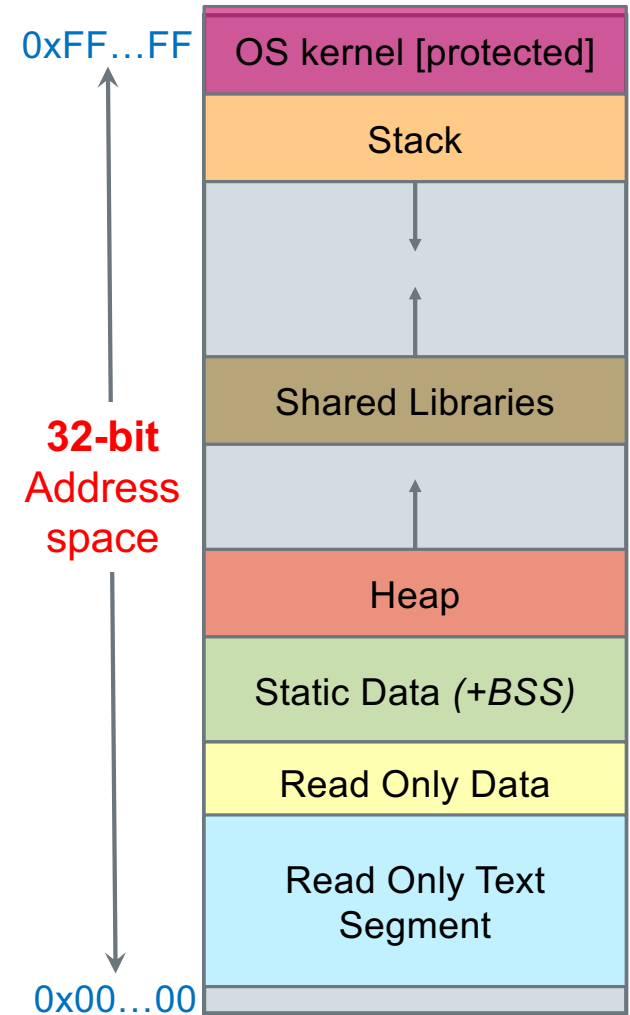
How to get a memory pointer into a register?

- Assembler **creates a table of pointers** in the **text segment** called the **literal table**
- For each variable in one of the data segments you reference in a special form of the ldr instruction (next slide), the assembler makes an entry for that variable whose contents is the 32-bit Label address

```
.bss
y: .space 4

.data
x: .word 200

.text
// your code
// last line of your code
// below is created by the assembler
.word y      // contents: 32-bit address of y
.word x      // contents: 32-bit address of x
```



Loading and using pointers in registers

- Tell the assembler to create and USE a literal table to obtain the address (Lvalue) of a label into a register:

```
ldr/str Rd, =Label // Rd = address
```

two step to **load** a **memory** variable

- load the pointer to the memory
- read (load) from the pointer

two steps **store** to a **memory** variable

- load the pointer to the memory
- write (store) to the pointer

```
.data
x: .word 200

.section .rodata
.Lmsg: .string "Hello World"

.text
// function header
main:

// load the contents into r2
ldr r2, =x // int *r2 = &x
ldr r2, [r2] // r2 = *r2;
// &x was only needed once above

// store the contents of r0
ldr r1, =.Lmsg // r1 = &y
str r0, [r1] // y = r0
// keeping &y in r1 above

...
```

How to use the literal table to get a big constant into a register

- In data processing instructions, the field **imm8 + rotate 4 bits** is too small to store many numbers outside of the range of -256 to 255, how do you get larger immediate values into a register?



fails



```
mov    r0, 1023
```

xxx.s:24: Error: invalid constant (3ff) after fixup

replacement



```
ldr     r0, =1023
```

- Answer: use **ldr** instruction with the constant as an operand: **=constant**
- Assembler creates a **literal table entry** with the **constant**

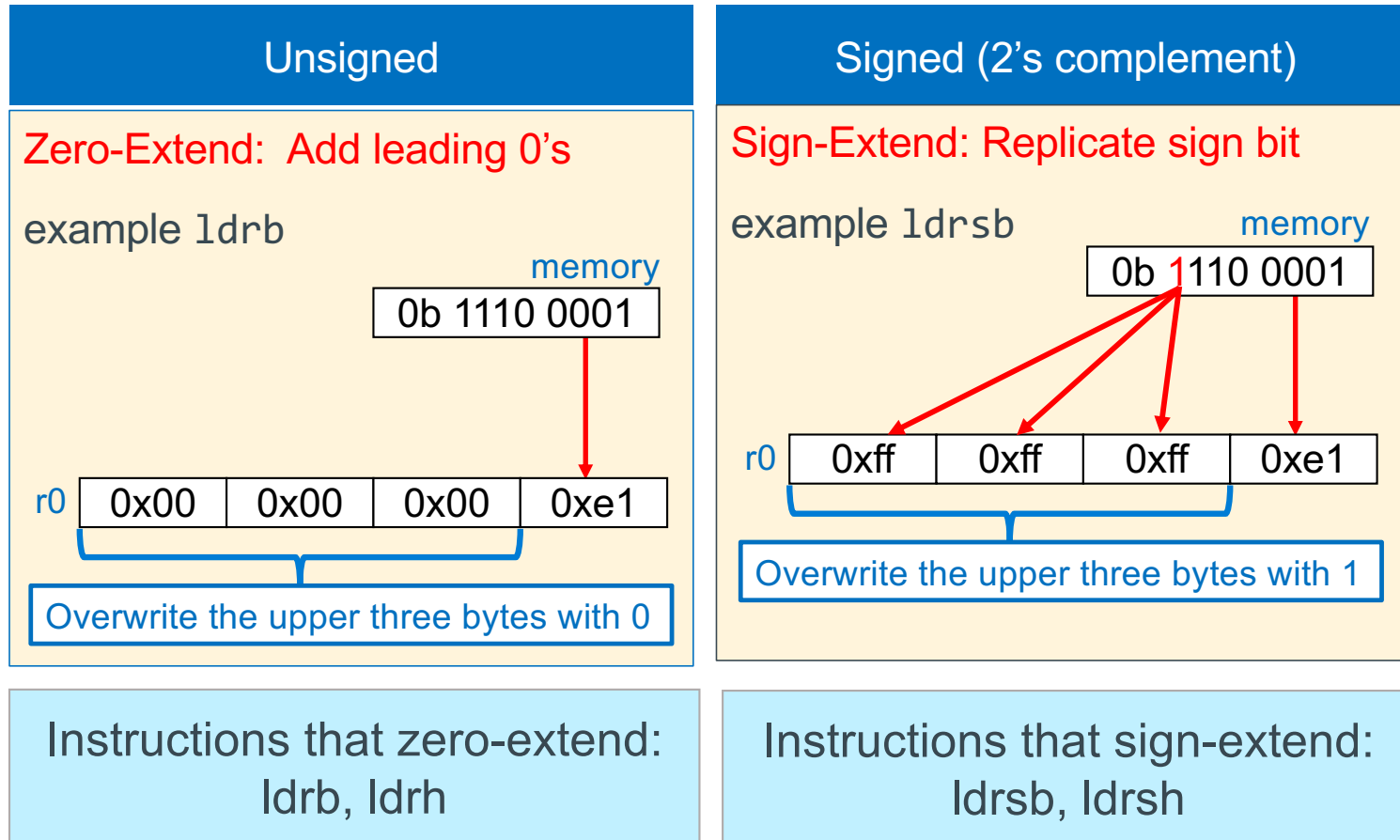
```
ldr  Rd, =constant    // =constant
ldr  r1, =0x2468abcd   // loads the constant 0x246abcd into r1
```

Loading and Storing: Variations List

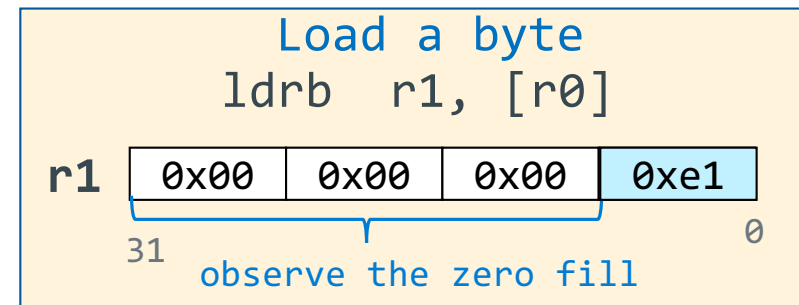
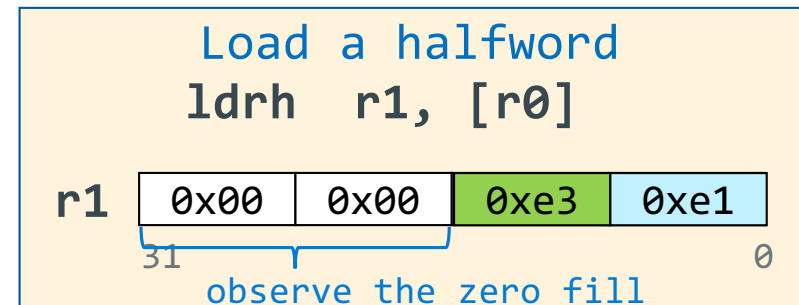
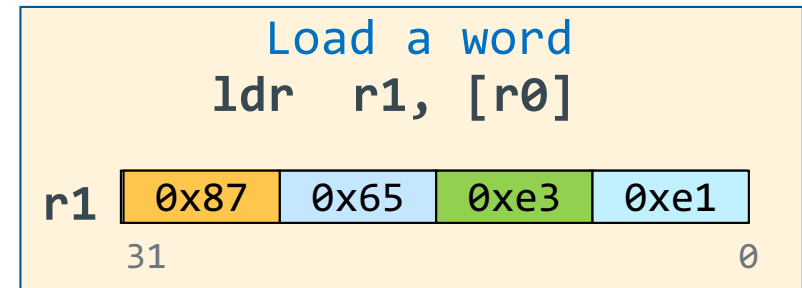
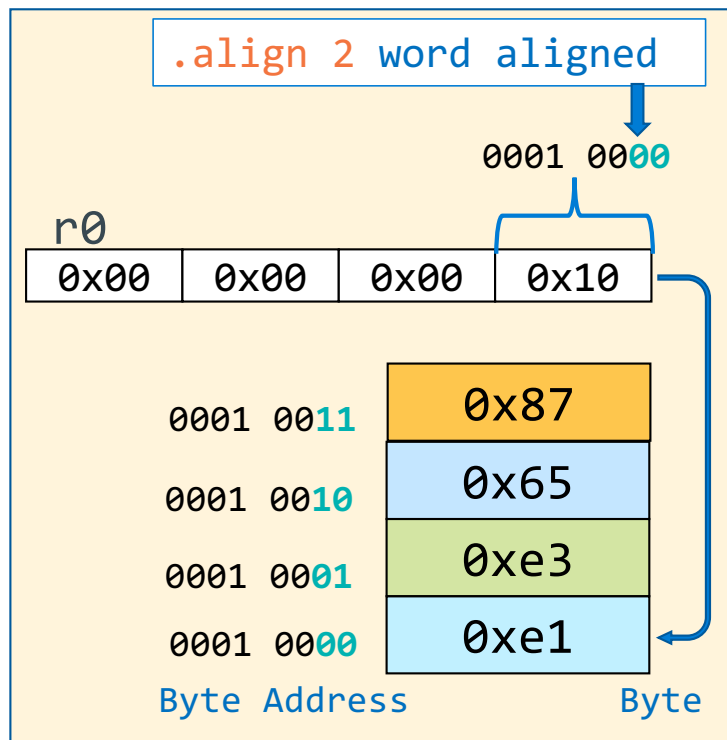
- Load and store have **variations** that move 8-bits, 16-bits and 32-bits
- Load into a register with less than 32-bits will **set the upper bits not filled from memory differently depending** on which **variation of the load instruction** is used
- Store will only select the lower 8-bit, lower 16-bits or all 32-bits of the register to copy to memory

Instruction	Meaning	Sign Extension	Memory Address Requirement
ldrsb	load signed byte	sign extension	none (any byte)
ldrb	load unsigned byte	zero fill (extension)	none (any byte)
ldrsh	load signed halfword	sign extension	halfword (2-byte aligned)
ldrh	load unsigned halfword	zero fill (extension)	halfword (2-byte aligned)
ldr	load word	---	word (4-byte aligned)
strb	store low byte (bits 0-7)	---	none (any byte)
strh	store halfword (bits 0-15)	---	halfword (2-byte aligned)
str	store word (bits 0-31)	---	word (4-byte aligned)

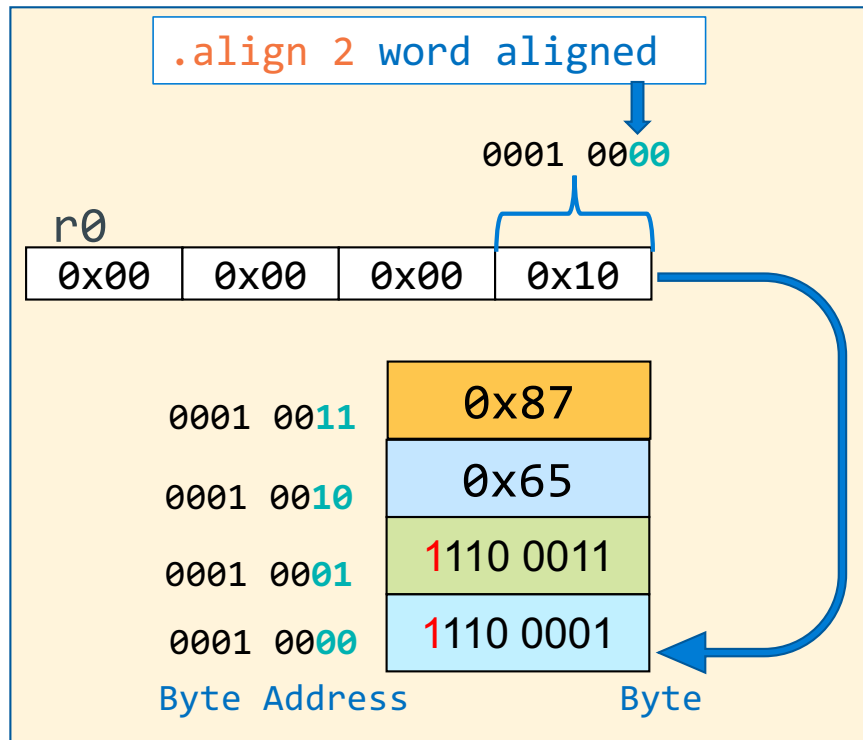
Loading 32-bit Registers From Memory Variables < 32-Bits Wide



Load a Byte, Half-word, Word

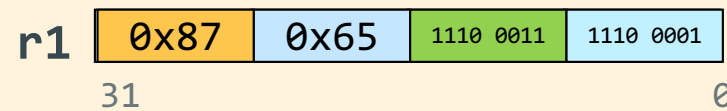


Signed Load a Byte, Half-word, Word



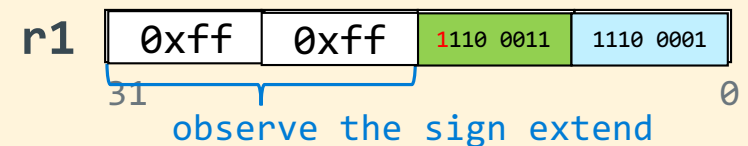
Load a word (no change)

```
ldr r1, [r0]
```



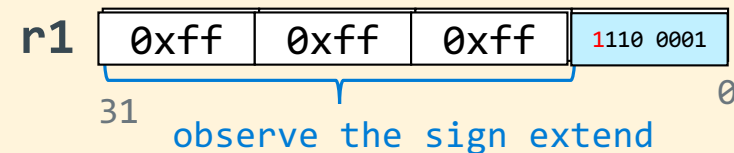
Load a halfword

```
ldrsh r1, [r0]
```

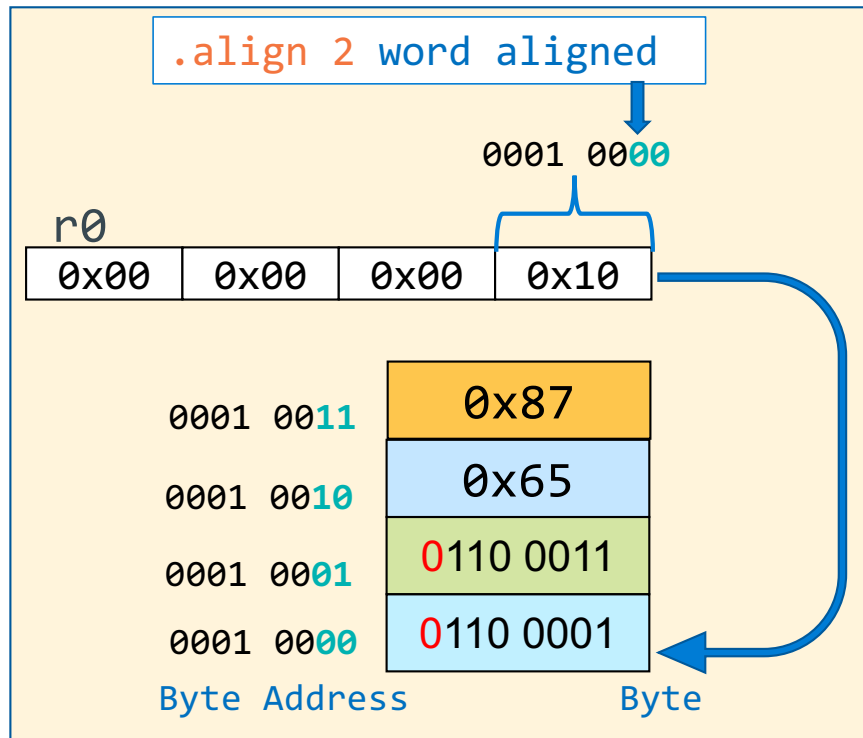


Load a byte

```
ldrshb r1, [r0]
```

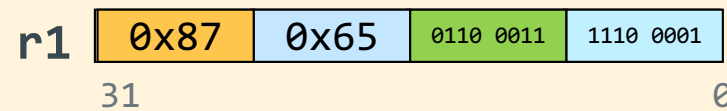


Signed Load a Byte, Half-word, Word



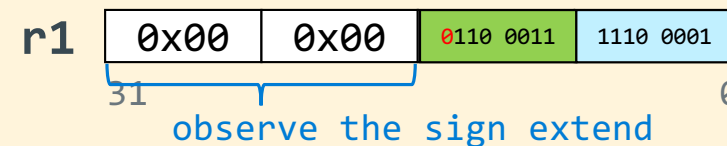
Load a word (no change)

ldr r1, [r0]



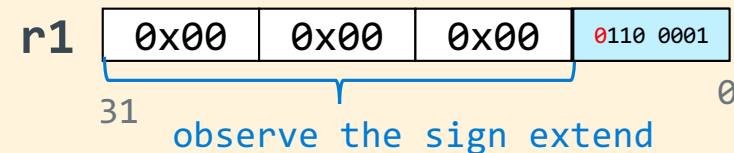
Load a halfword

ldrsh r1, [r0]

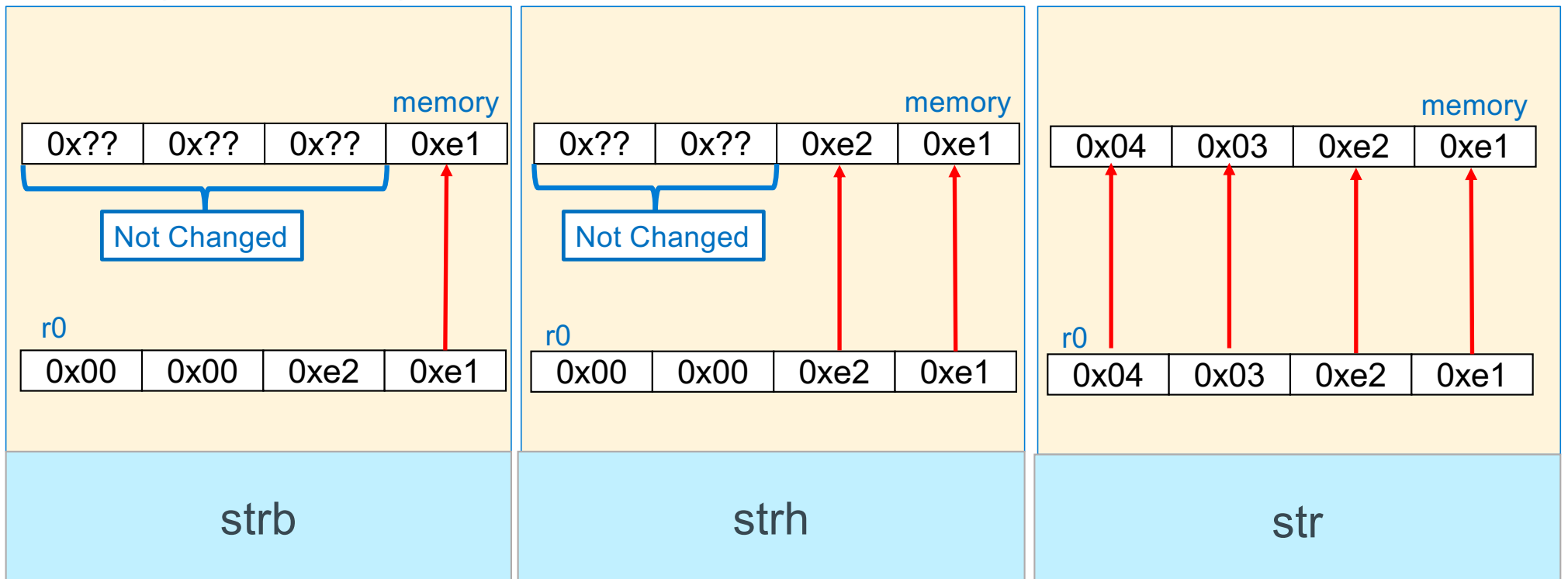


Load a byte

ldrshb r1, [r0]



Storing 32-bit Registers To Memory 8-bit, 16-bit, 32-bit



Store a Byte, Half-word, Word

initial value in r0

0x20	0x00	0x00	0x00
------	------	------	------

Store a byte
`strb r1, [r0]`

r1: 31 0x87 0x65 0xe3 0xe1 0

Byte Address	Byte
0x20000003	0x33
0x20000002	0x22
0x20000001	0x11
0x20000000	0xe1

observe other bytes NOT altered

Store a halfword
`strh r1, [r0]`

r1: 31 0x87 0x65 0xe3 0xe1 0

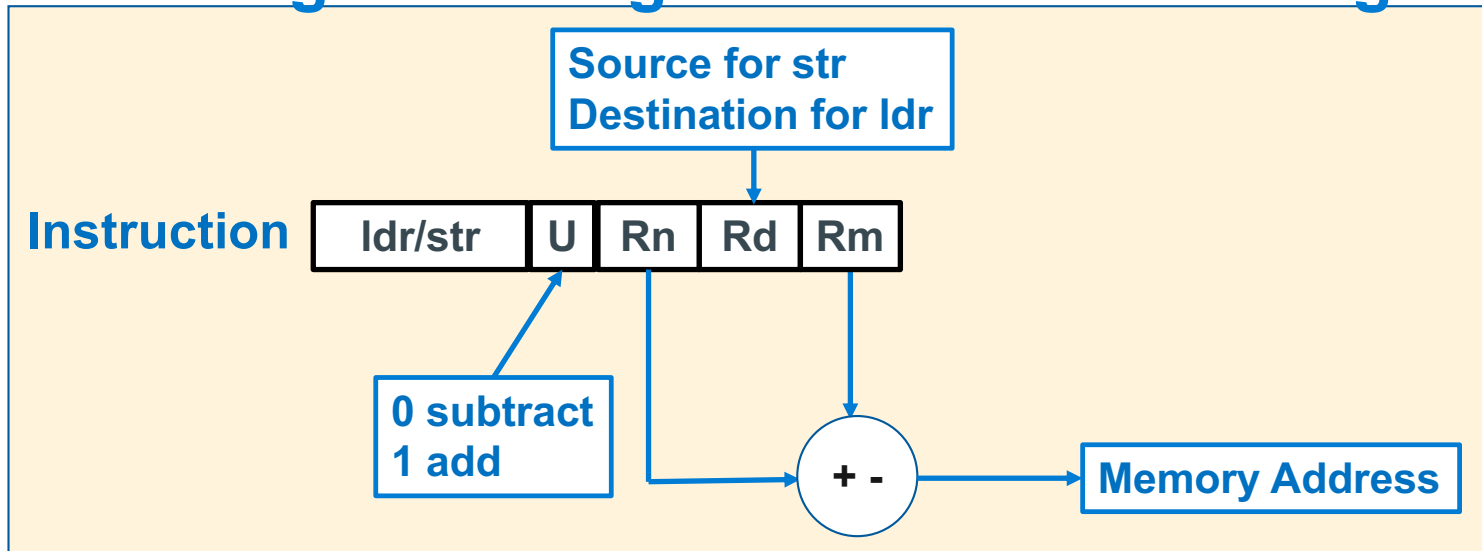
Byte Address	Byte
0x20000003	0x33
0x20000002	0x22
0x20000001	0xe3
0x20000000	0xe1

Store a word
`str r1, [r0]`

r1: 31 0x87 0x65 0xe3 0xe1 0

Byte Address	Byte
0x20000003	0x87
0x20000002	0x65
0x20000001	0xe3
0x20000000	0xe1

ldr/str Base Register + Register Offset Addressing



Pointer Address = Base Register + Register Offset

- **Unsigned** offset integer **in a register (bytes)** is either added/subtracted from the **pointer address** in the **base register**

Syntax	Address	Examples
<code>ldr/str Rd, [Rn +/- Rm]</code>	$Rn + \text{ or } - Rm$	<code>ldr r0, [r5, r4]</code> <code>str r1, [r5, r4]</code>

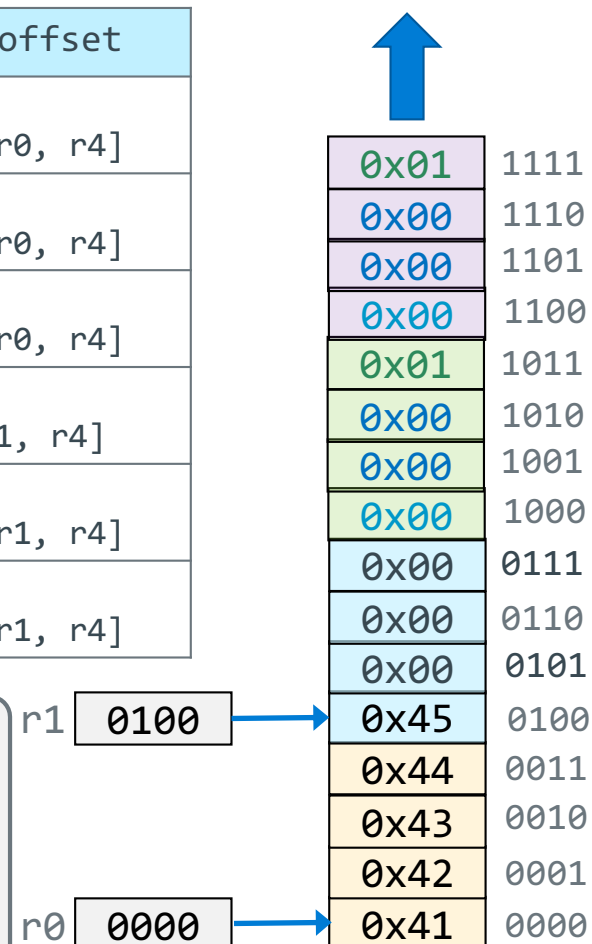
Array addressing with ldr/str

Array element	Base addressing	Immediate offset	register offset
ch[0]	ldrb r2, [r0]	ldrb r2, [r0, 0]	mov r4, 0 ldrb r2, [r0, r4]
ch[1]	add r0, r0, 1 ldrb r2, [r0]	ldrb r2, [r0, 1]	mov r4, 1 ldrb r2, [r0, r4]
ch[2]	add r0, r0, 2 ldrb r2, [r0]	ldrb r2, [r0, 2]	mov r4, 2 ldrb r2, [r0, r4]
x[0]	ldr r2, [r1]	ldr r2, [r1, 0]	mov r4, 0 ldr r2, [r1, r4]
x[1]	add r1, r1, 4 ldrb r2, [r1]	ldrb r2, [r1, 4]	mov r4, 4 ldrb r2, [r1, r4]
x[2]	add r1, r1, 8 ldrb r2, [r1]	ldrb r2, [r1, 8]	mov r4, 8 ldrb r2, [r1, r4]

table rows are
independent instructions

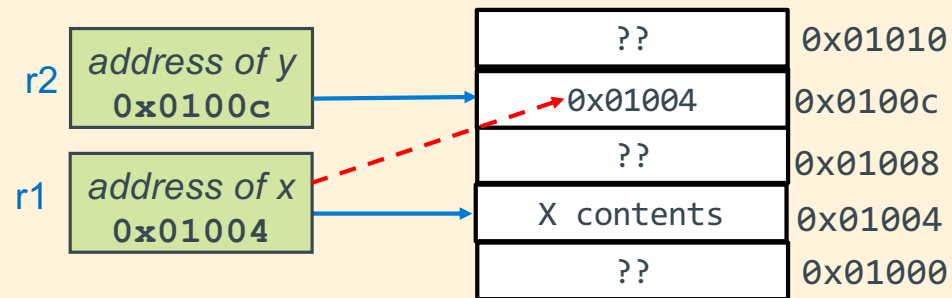
```

.data
ch:  .byte 0x41, 0x42, 0x43, 0x44
x:   .word 0x00000045
     .word 0x01000000
     .word 0x01020304
.text
ldr  r0, =ch
ldr  r1, =x
    
```



ldr/str practice - 1

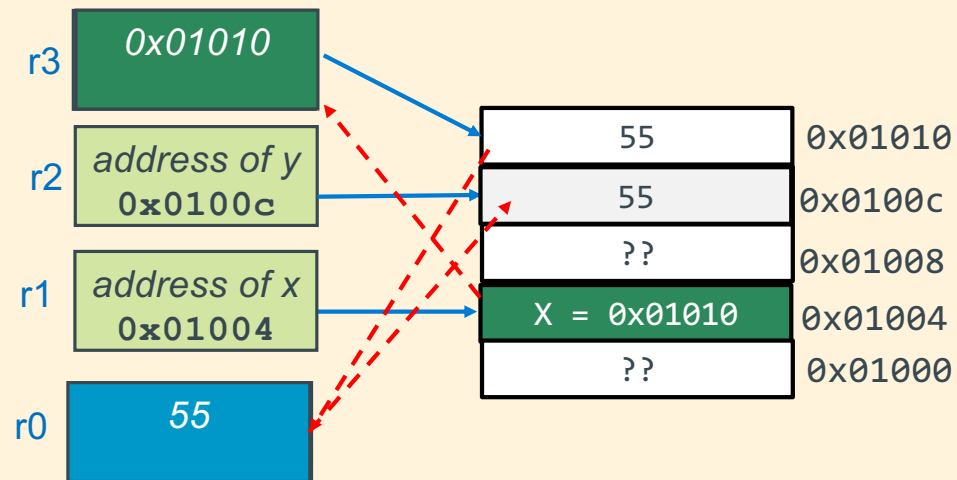
r1 contains the Address of X (defined as int X) in memory; r1 points at X
r2 contains the Address of Y (defined as int *Y) in memory; r2 points at Y
write Y = &X;



```
str    r1, [r2]    // y ← &x
```

ldr/str practice - 2

r1 contains the Address of X (defined as `int *X`) in memory r1 points at X
r2 contains the Address of Y (defined as `int Y`) in memory; r2 points at Y
write `Y = *X;`



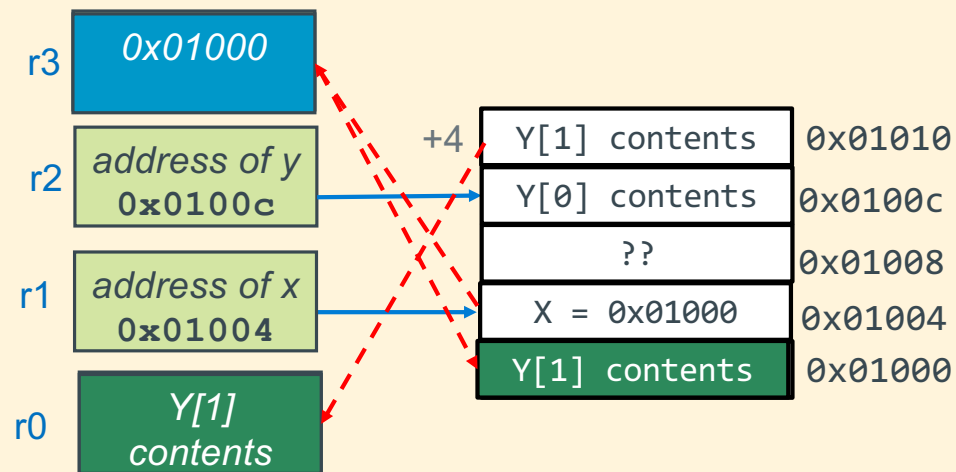
```
ldr    r3, [r1]    // r3 ← x (read 1)
ldr    r0, [r3]    // r0 ← *x (read 2)
str    r0, [r2]    // y ← *x
```

ldr/str practice - 3

r1 contains Address of X (defined as `int *X`) in memory; r1 points at X

r2 contains Address of Y (defined as `int Y[2]`) in memory; r2 points at `&(Y[0])`

`write *X = Y[1];`



```
ldr    r0, [r2, 4]    // r0 ← y[1]
ldr    r3, [r1]        // r3 ← x
str     r0, [r3]        // *x ← y[1]
```

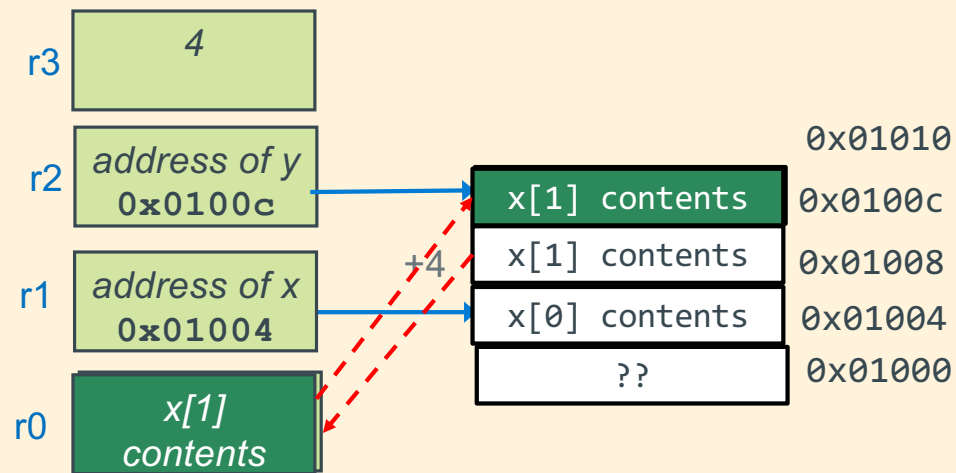
ldr/str practice - 4

r1 contains Address of X (defined as `int X[2]`) in memory; r1 points at `&(x[0])`

r2 contains Address of Y (defined as `int Y`) in memory; r2 points at Y

r3 contains a 4

write `Y = X[1];`



```
ldr    r0, [r1, r3] // r0 ← x[1]
```

```
str    r0, [r2]     // y ← x[1]
```


Label (Address) Math

- You can have the assembler calculate some useful values for you
- One common use is calculating the distance in bytes between two labels
- The dot (.) refers to the address on the current line (the next byte after a previous space allocation)

```
.section .rodata
.Lst: .string "The value of x is %d\n"
.equ STSZ, (. - .Lst)    // number of bytes in .Lst includes \0
.equ STLEN, STSZ - 1     // string length of .Lst
```

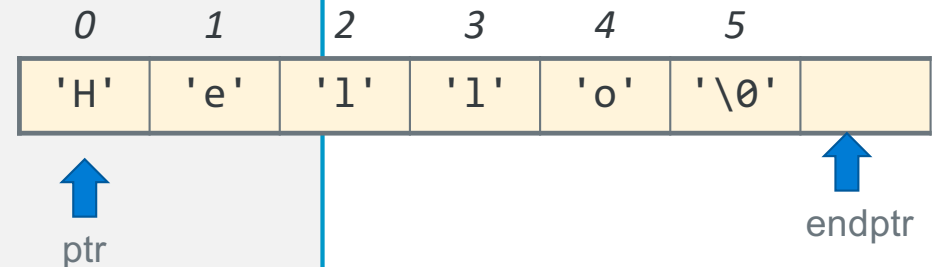
Example: Base Register Addressing with Arrays

```
#include <stdio.h>
#include <stdlib.h>

char msg[] = "Hello CSE30! We Are CountinG UpPER cASe letters!";

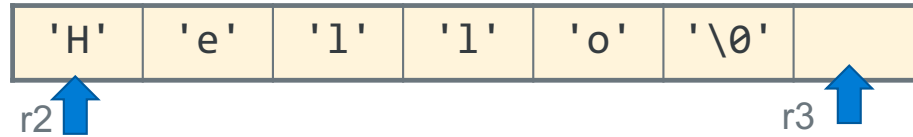
int
main(void)
{
    int cnt = 0;
    char *endpt = msg + sizeof(msg)/sizeof(*msg);
    char *ptr = msg;

    while(ptr < endpt) {
        if ((*ptr >= 'A') && (*ptr <= 'Z'))
            cnt++;
        ptr++;
    }
    printf("%d\n", cnt);
    return EXIT_SUCCESS;
}
```



Example: Base Register Addressing with Arrays

- Iterates a pointer (r2) through the array
- r3 contains the address +1 past the end of the string
- MSGSZ is the size of the array (including the '\0') if you wanted to excluded the '\0', then subtract 1 from MSGSZ
- Use **ldrb** as **msg** is an array of chars



```

.data          // segment
msg:.string    "Hello CSE30! We Are CountinG UpPER cASe letters!"
.equ          MSGSZ, (. - msg) // number of bytes in msg
.section .rodata
.Lpf:.string   "%d\n"          // literal for printf
...
    ldr        r2, =msg        // ptr point to &msg
    add        r3, r2, MSGSZ   // endpt points after end

.Lwhile:
    cmp        r2, r3          // at end of buffer yet?
    bge        .Lexit          loop guard

    ldrb       r0, [r2]        // get next char (base addressing)
    cmp        r0, 'A'         // is it less than an 'A' ?
    blt        .Lendif         // if so, not CAP (short circuit)
    cmp        r0, 'Z'         // is it greater than a 'Z'?
    bgt        .Lendif         // if so, not CAP
    add        r1, r1, 1       // is a CAP increment

.Lendif:
    add        r2, r2, 1       // move to next char
    b          .Lwhile         // go to loop guard at top of while
.Lexit:

```

Example: Base Register + Offset Register

```
ldr    r2, =msg           // ptr point to &msg
add    r3, r2, MSGSZ      // endpt points after end
.Lwhile:
cmp    r2, r3             // at end of buffer yet?
bge    .Lexit

ldrb   r0, [r2]           // get next char
cmp    r0, 'A'            // is it less than an 'A' ?
blt    .Lendif            // if so, not CAP
cmp    r0, 'Z'            // is it greater than a 'Z'?
bgt    .Lendif            // if so, not CAP
add    r1, r1, 1          // is a CAP increment

.Lendif:
add    r2, r2, 1          // move to next char
b      .Lwhile            //go to loop guard while top
.Lexit:
```

Using Base register pointer with an end pointer

```
ldr    r2, =msg           // ptr point to &msg
mov    r3, 0              // index reg
.Lwhile:
cmp    r3, MSGSZ          // are we done?
bge    .Lexit

ldrb   r0, [r2, r3]       // get next char
cmp    r0, 'A'            // is it less than an 'A' ?
blt    .Lendif            // if so, not CAP
cmp    r0, 'Z'            // is it greater than a 'Z'?
bgt    .Lendif            // if so, not CAP
add    r1, r1, 1          // is a CAP increment

.Lendif:
add    r3, r3, 1          // index++
b      .Lwhile
.Lexit:
```

Using Base register pointer + Offset register

Example: Base Register + Register Offset Two Buffers

```
#include <stdio.h>
#include <stdlib.h>
#define SZ 6

int src[SZ] = {1, 3, 5, 7, 9, 11};

int dest[SZ];
int
main(void)
{
    for (int i = 0; i < SZ; i++)
        dest[i] = src[i];

    return EXIT_SUCCESS;
}
```

- Make sure to index by bytes and increment the index register by `sizeof(int) = 4`

```
.data                // segment
src:.word            1, 3, 5, 7, 9, 11
.equ                SRCSZ, (. - src) // bytes msg
dest:.space          SRCSZ
.equ                INT_STEP, 4
...

ldr    r0, =src      // ptr to src
ldr    r1, =dest     // ptr to dest
mov    r2, 0

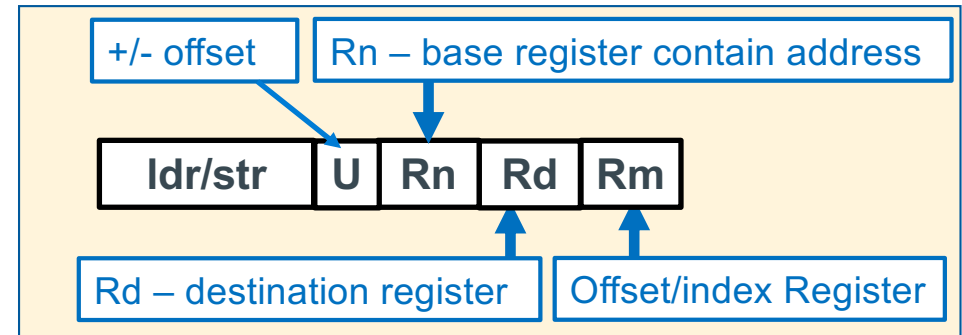
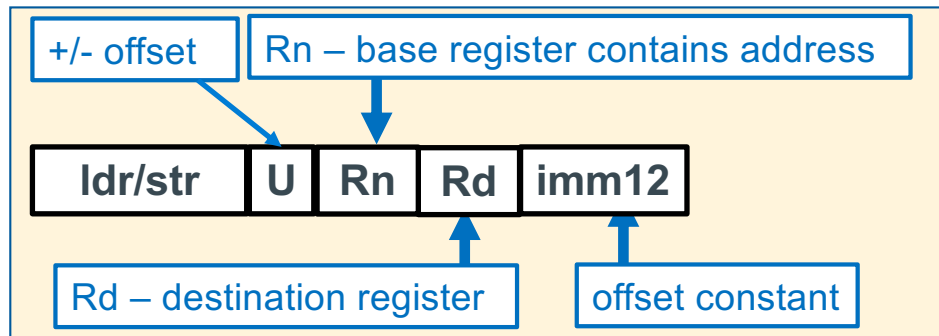
.Lfor:
    cmp    r2, SRCSZ // in bytes!
    bge    .Lexit

    ldr    r3, [r0, r2]
    str    r3, [r1, r2]
    add    r2, r2, INT_STEP
    b     .Lfor

.Lexit:
```

one increment
covers both arrays

Reference: LDR/STR – Register To/From Memory Copy



```
ldr/str Rd, [Rn, +/- imm12] // base register pointer + offset  imm12 in bytes
                             -4095 <= imm12 <= 4095 (bytes)
ldr/str Rd, [Rn]             // base register pointer + 0 (imm12 is 0)
ldr/str Rd, [Rn, +/- Rm]     // base register pointer +/- offset register
```

```
ldr      r1, =var_x           // r1 = &var_x
str      r1, =mylabel+4       // *(mylabel+4) = r1
ldr      r1, =0x246abcd       // load an immediate into r1
ldr      r1, [r3]             // y = *r3 (4 bytes)
str      r1, [r0]             // *r0 = r1
ldr      r1, [r3, -4]         // y = *(r3 - 4) (4 bytes)
str      r1, [r0, r2]         // *(r0 + r2) = r1
```

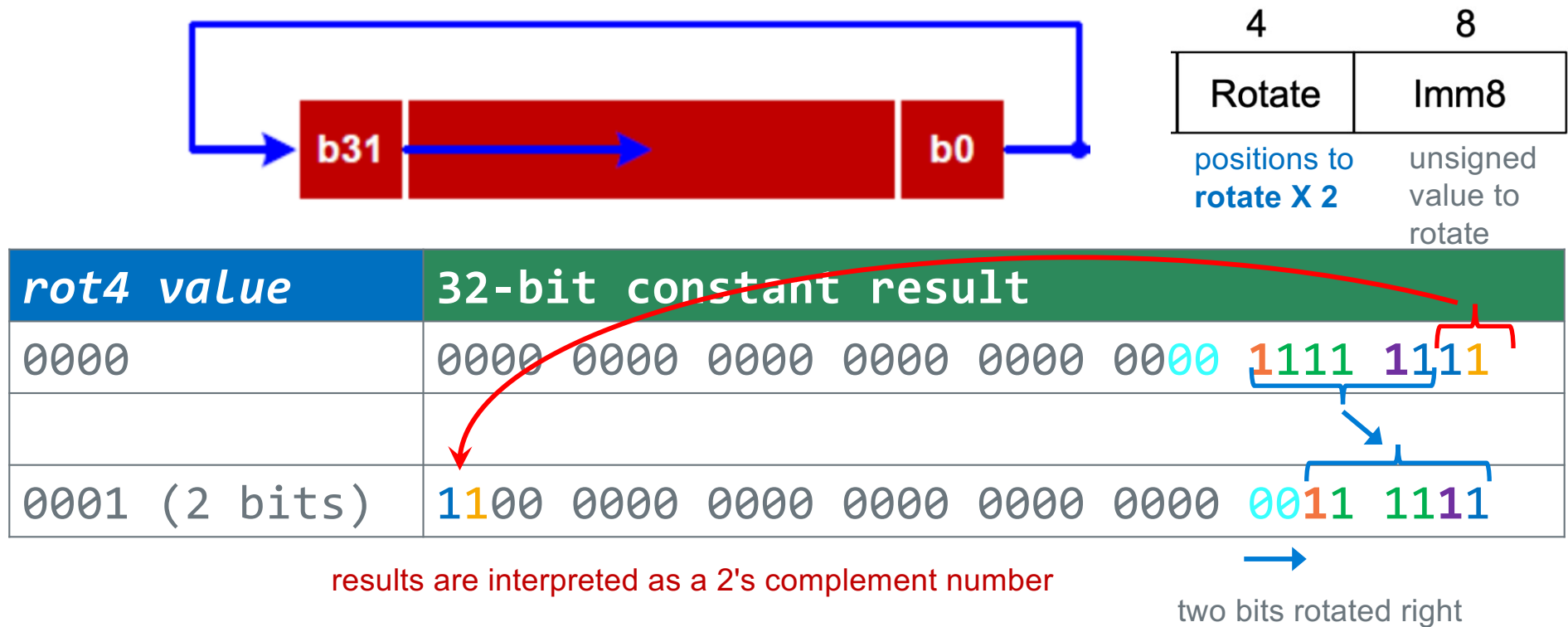
Reference: Addressing Mode Summary for use in CSE30

index Type	Example	Description
Pre-index immediate	<code>ldr r1, [r0]</code>	$r1 \leftarrow \text{memory}[r0]$ r0 is unchanged
Pre-index immediate	<code>ldr r1, [r0, 4]</code>	$r1 \leftarrow \text{memory}[r0 + 4]$ r0 is unchanged
Pre-index immediate	<code>str r1, [r0]</code>	$\text{memory}[r0] \leftarrow r1$ r0 is unchanged
Pre-index immediate	<code>str r1, [r0, 4]</code>	$\text{memory}[r0 + 4] \leftarrow r1$ r0 is unchanged
Pre-index register	<code>ldr r1, [r0, +-r2]</code>	$r1 \leftarrow \text{memory}[r0 \pm r2]$ r0 is unchanged
Pre-index register	<code>str r1, [r0, +-r2]</code>	$\text{memory}[r0 \pm r2] \leftarrow r1$ r0 is unchanged

Extra Slides

How are I – Type Constants Encoded in the instruction?

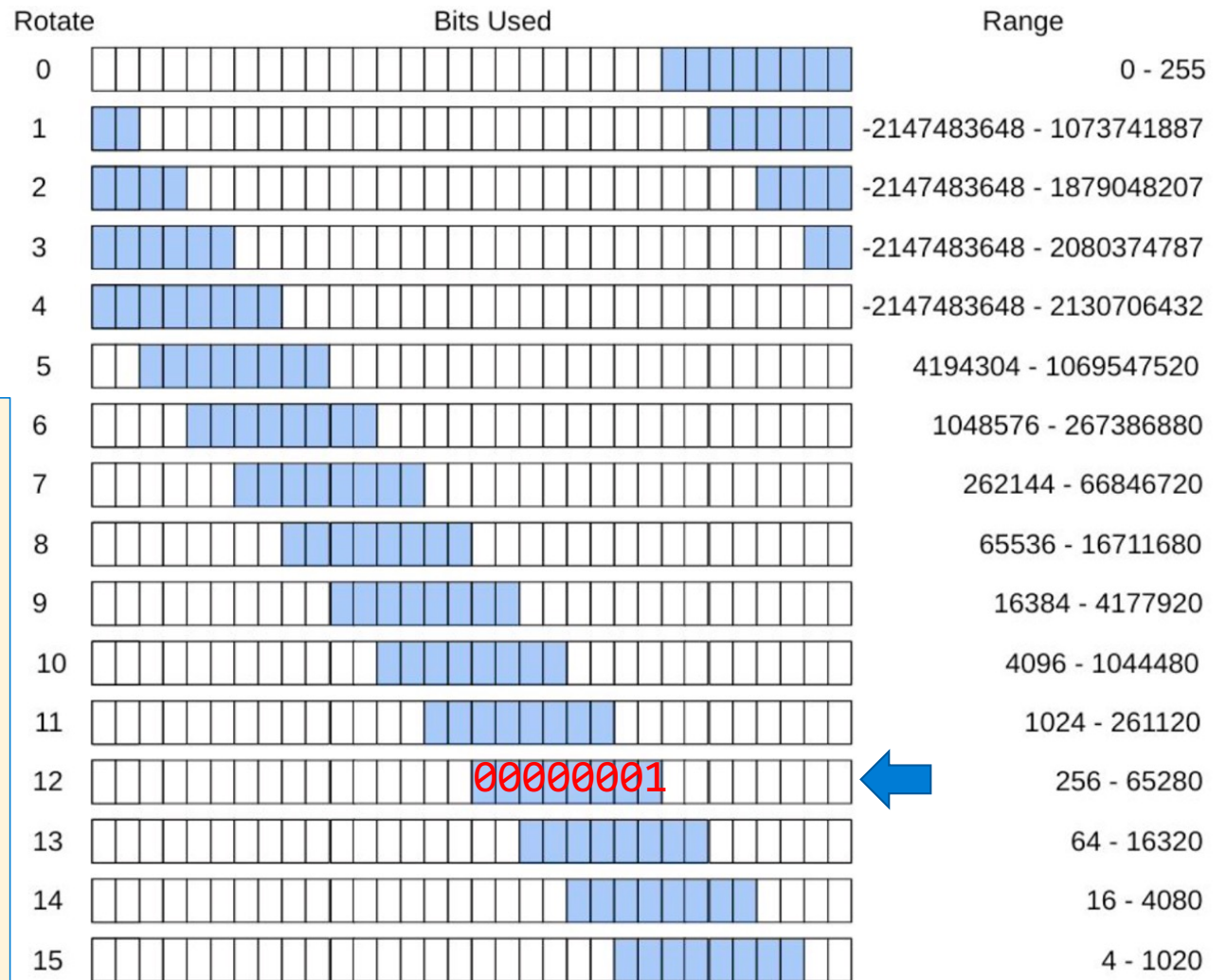
- Aarch32 provides only 8-bits for specifying an immediate constant value
- Without "rotation" immediate values are limited to the range of positive 0-255
- Imm8 expands to 32 bits and does a rotate right to achieve additional constant values (YUCK)



Rot4 - Imm8 Values

4	8
Rotate	Imm8
positions to rotate X 2	unsigned value to rotate

- 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



results are interpreted as a 2's complement number

x

Branch Target Address (BTA): What Is imm24?

- Previous slide: **phases of execution:**
(1) fetch, (2) decode, (3) execute
- The pc (r15) contains the address of the **instruction being fetched**, which is two instructions ahead or **executing instruction + 8 bytes**
- **Branch target address** (or imm24) is the **distance measured** in the **# of instructions** (signed, 2's complement) from the **fetch address** contained in **r15** when executing the branch

executing instruction

decode instruction

fetch instruction

```

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: 1affffff5     bne 1042c <inloop>
    
```

BTA: + 2 instructions

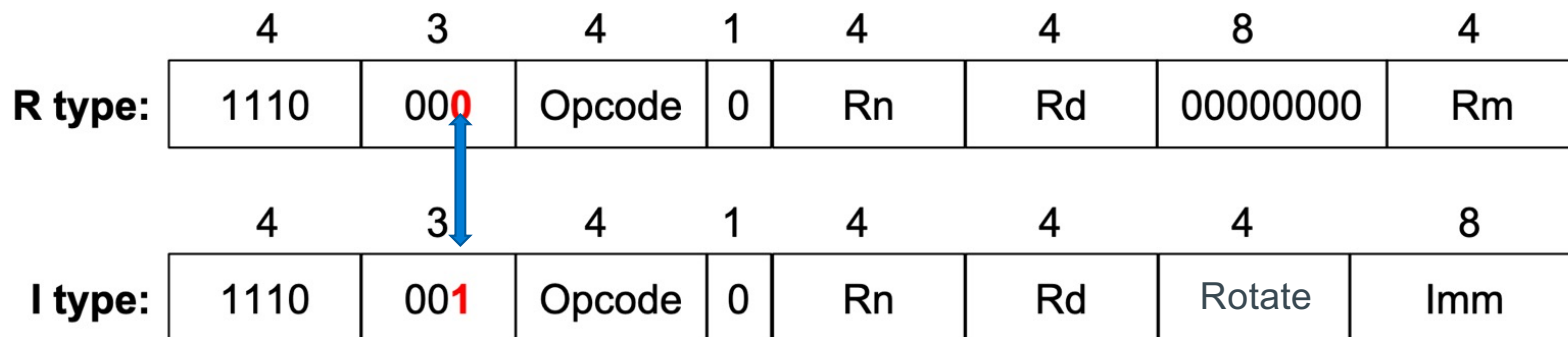
```

target address    = 0x10440
fetch address     = 0x10438
distance(bytes)   = 0x00008
distance(instructions) = 0x8/(4 bytes/instruction)= 0x2
    
```

imm24	0x 00 00 02
-------	-------------

Basic Arm Machine Code 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)
- R type instruction: `add r0, r1, r2` // third operand is a register
- I type instruction: `add r0, r0, 1` // third operand is an immediate value



Literal Table (Array) each entry is a pointer to a different Label

- Assembler automatically inserts into the text segment an array (table) of pointers
- Each entry contains a 32-bit address of one of the labels
- Uses r15 (PC) as base register to load the entry into a reg

$\text{displacement (bytes)} - 8$

The assembler creates this table before generating the .o file

```
.bss
y: .space 4

.data
x: .word 200

.section .rodata
.Lmsg: .string "Hello World"

.text
main:
(address)ldr r0, [PC, displacement] // replaces: ldr r0, =y
    <last line of your assembly, typically a function return>

.word y      // entry #1 32-bit address for y
.word x      // entry #2 32-bit address for x
.word .Lmsg  // entry #3 32-bit address for .Lmsg
```

Literal Table (Array) each entry is a pointer to a different Label

The displacement is different for each use. As the PC is different at each instruction

```
.bss
y: .space 4
.data
x: .word 200
.section .rodata
.Lmsg: .string "Hello World"
.text
main:
(address)ldr r0, [PC, displacement1] // replaces: ldr r0, =y
(address)ldr r0, [PC, displacement2] // replaces: ldr r0, =y
<last line of your assembly, typically a function return>
.word y // entry #1 32-bit address for y
.word x // entry #2 32-bit address for x
.word .Lmsg // entry #3 32-bit address for .Lmsg
```

displacement1 - 8

displacement2 - 8

ARM Assembly Source File: Header

File Header

At the top of every
ARM source file

```
.arch    armv6           // armv6 architecture
.arm     // arm 32-bit instruction set
.fpu     vfp             // floating point co-processor
.syntax  unified         // modern syntax
```

```
// Contents of the other memory segment include .text (your code)
```

.arch <architecture>

- Specifies the target architecture to generate machine code
- Typically specify oldest ARM arch you want the code to run on – most arm CPUs are backwards compatible

.arm

- Use the 32-bit ARM instructions, There is an alternative 16-bit instruction set called thumb that we will not be using

.fpu <version>

- Specify which floating point co-processor instructions to use (OPTIONAL we will not be using floating point)

ARM Assembly Source File: Header and Footer

File Header

At the top of every ARM source file

```
.arch    armv6           // armv6 architecture
.arm     // arm 32-bit instruction set
.fpu     vfp             // floating point co-processor
.syntax  unified         // modern syntax
```

```
// Contents of the other memory segment include .text (your code)
```

File Footer

At the bottom of every ARM source file

```
.section .note.GNU-stack,"",%progbits // set stack/data non-exec
.end

// everything past the .end is ignored!
// Debugging notes etc
```

`.syntax unified`

- use the standard ARM assembly language syntax called *Unified Assembler Language (UAL)*

`.section .note.GNU-stack,"",%progbits`

- tells the linker to **make the stack and all data segments not-executable** (no instructions in those sections) – security measure

`.end`

- at the end of the source file, everything written after the `.end` is ignored

Function Header and Footer Assembler Directives

function entry point
address of the first
instruction in the function
Must not be a local label
(does not start with .L)

```
        .text
Function Header {
    .global myfunc           // make myfunc global for linking
    .type    myfunc, %function // define myfunc to be a function
    .equ     FP_OFF, 4       // fp offset in main stack frame
myfunc:
    // function prologue, stack frame setup
    // your code
    // function epilogue, stack frame teardown
Function Footer {
    .size myfunc, (. - myfunc)
```

.global function_name

- Exports the function name to other files. Required for main function, optional for others

.type name, %function

- The **.type** directive sets the **type of a symbol/label name**
- %function** specifies that **name** is a function (name is the address of the first instruction)

equ FP_OFF, 4

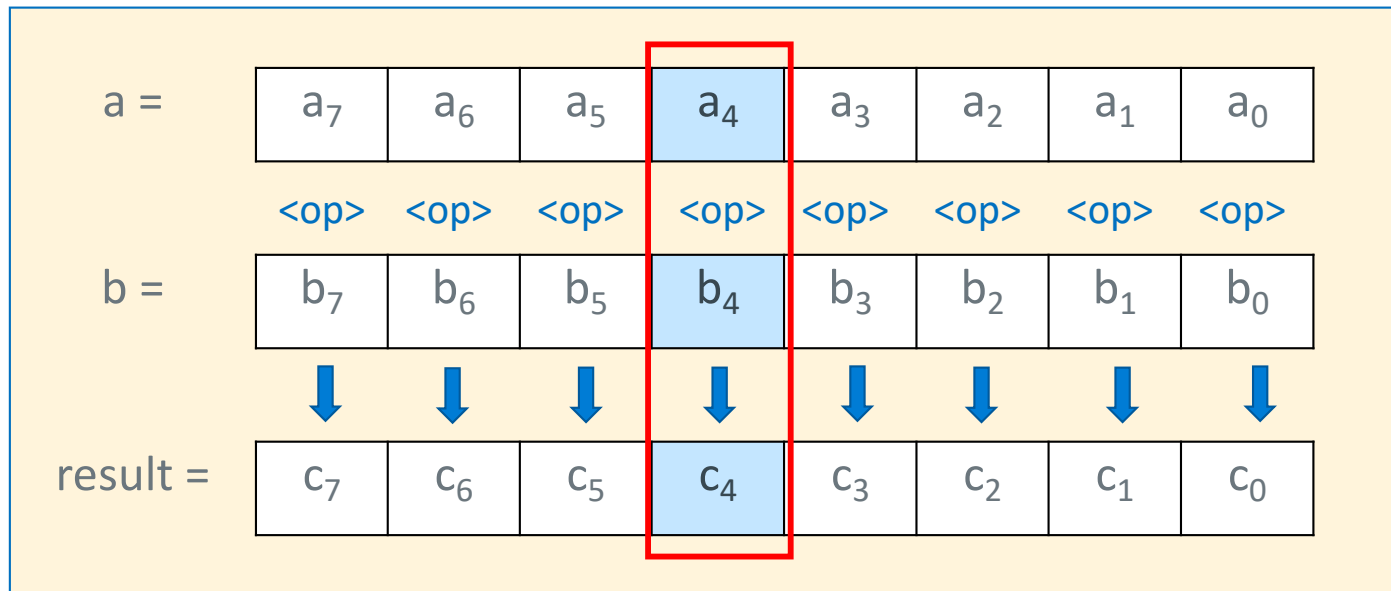
- Used for basic stack frame setup; the number 4 will change – later slides

.size name, bytes

- The **.size** directive is used to **set the size associated with a symbol**
- Used by the linker to exclude unneeded code and/or data when creating an executable file
- It is also used by the **debugger** gdb
- bytes is best calculated as an expression: (period is the current address in a memory segment)**

In CSE30 required use: .size name, (. - name)

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

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

First Look: Copying Values To Registers – MVN (not)

mvn r0, r1

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

register r1



register r0

mvn r0, 12

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

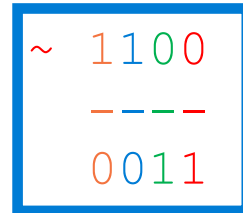
register r0

0x0c



0xffff fff3

Bitwise NOT



- A **bitwise NOT** operation

0x 0c

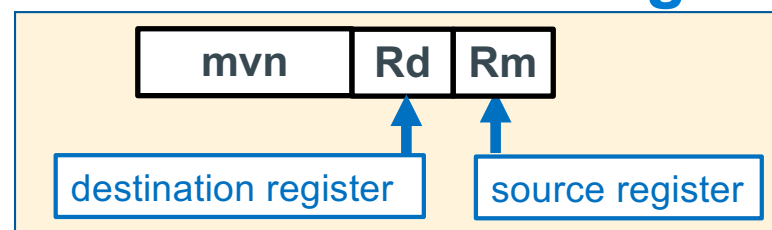
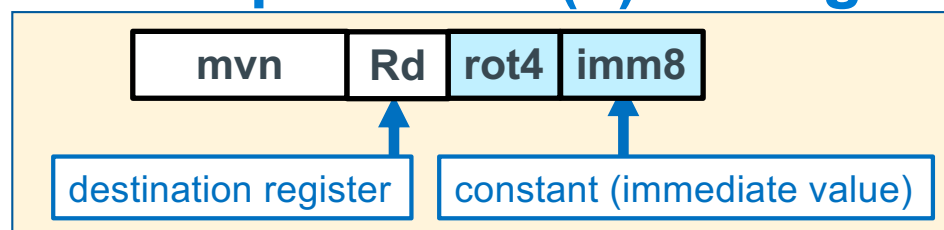
↓ imm8 expansion

0x0000000c

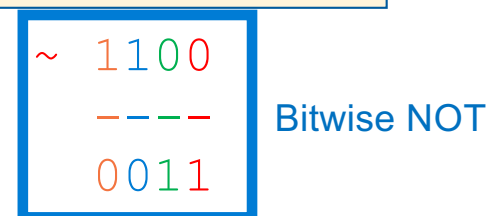
↓ bitwise not

0xfffffffff3

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



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



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	r1	0xffffffffb	←	0x00000004	←	0x4
mvn	r1, r5	// x = ~y in C	r1	0x55555555	←	0xaaaaaaaa	←	r5
mvn	r1, 0	// x = -1	r1	0x11111111	←	0x0	←	

Labels above the first row: invert the bits, copy into 32 bits zero extend