

Version 2.00

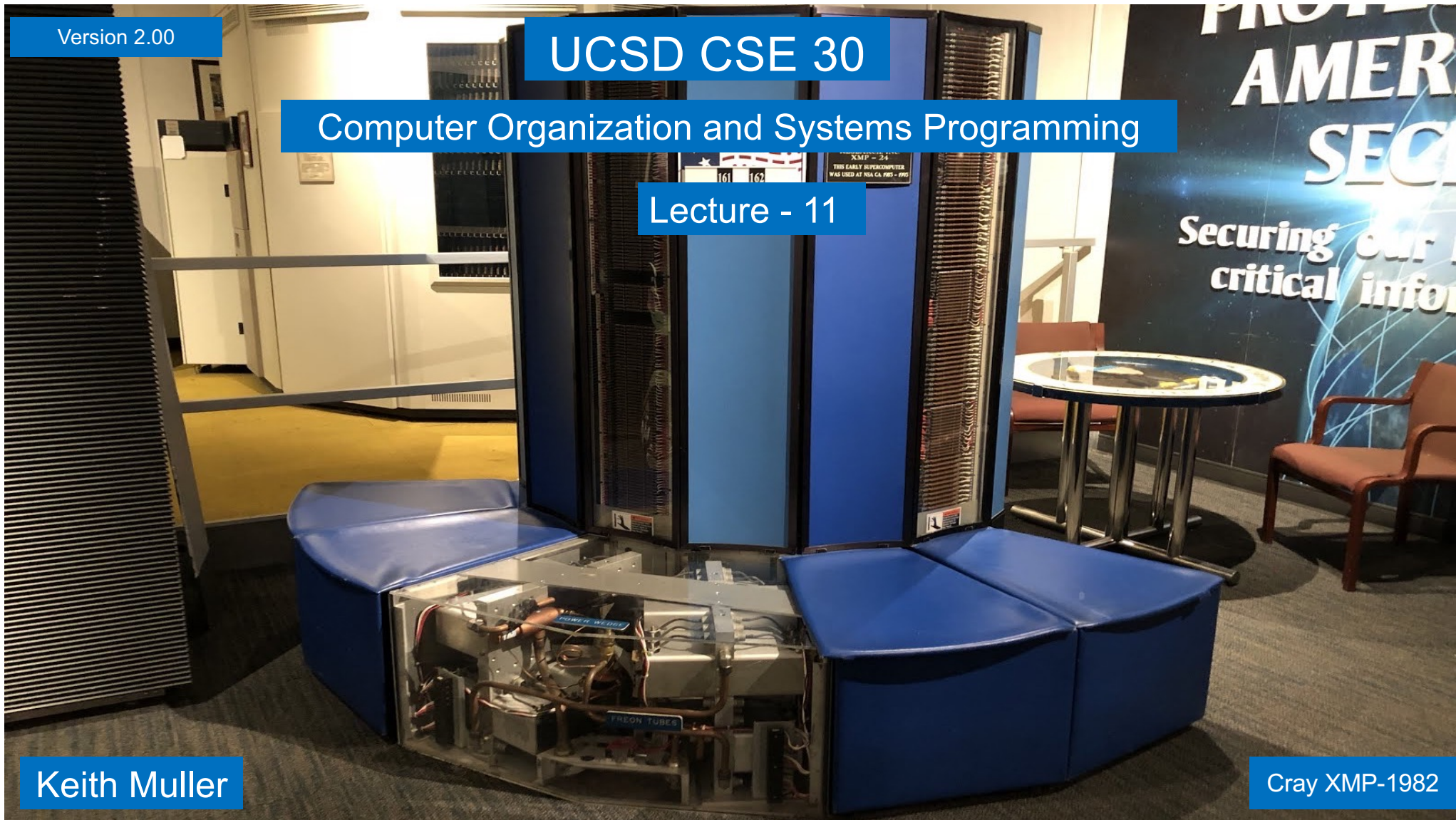
UCSD CSE 30

Computer Organization and Systems Programming

Lecture - 11

Keith Muller

Cray XMP-1982





Creating a Node & Inserting it at the Front of the List

```
// create node; insert at front when passed head
struct node *
creatNode(int year, char *name, struct node *link)
{
    struct node *ptr = malloc(sizeof(*ptr));
    if (ptr != NULL) {
        if ((ptr->name = strdup(name)) == NULL) {
            free(ptr);
            return NULL;
        }
        ptr->year = year;
        ptr->next = link;
    }
    return ptr;
}
```

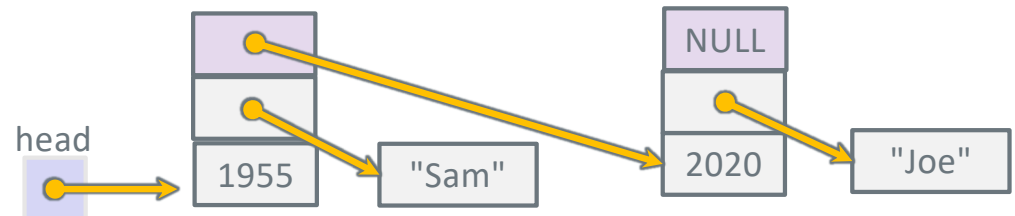
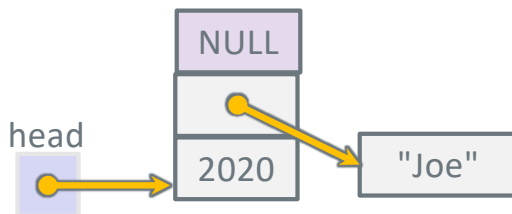
Never use head here!
you can lose your linked list
if creatNode() fails!

```
struct node {
    int year;
    char *name;
    struct node *next;
};
```

```
// calling function body
struct node *head = NULL; // insert at front
struct node *ptr;

if ((ptr = creatNode(2020, "Joe", head)) != NULL) {
    head = ptr; // error handling not shown
}
if ((ptr = creatNode(1955, "Sam", head)) != NULL) {
    head = ptr; // error handling not shown
}
```

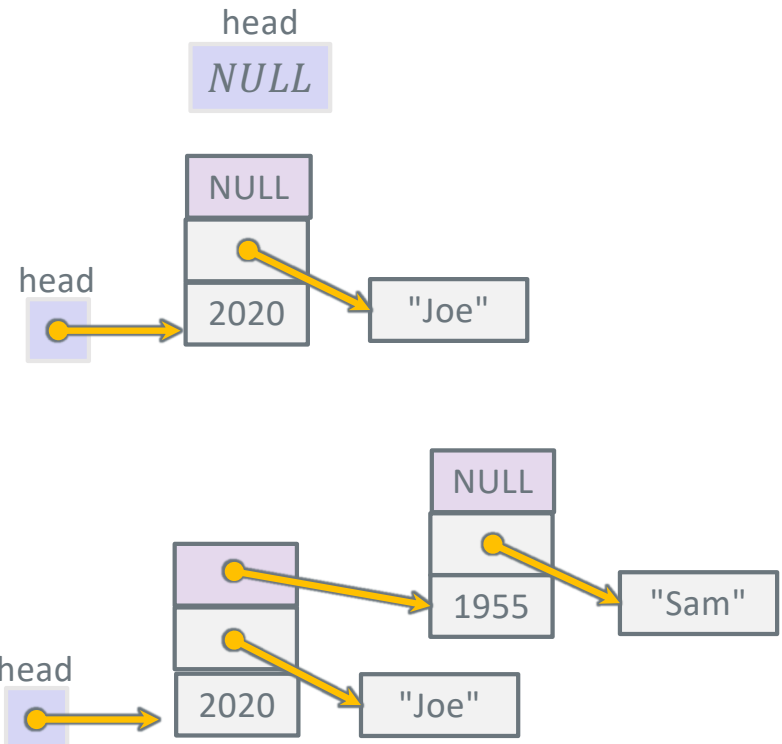
head
NULL



Creating a Node & Inserting it at the **End** of the List

// create a node and insert at the end of the list

```
struct node *  
insertEnd(int year, char *name, struct node *head)  
{  
    struct node *ptr = head;  
    struct node *prev = NULL; // base case  
    struct node *new;  
  
    if ((new = creatNode(year, name, NULL)) == NULL)  
        return NULL;  
  
    while (ptr != NULL) {  
        prev = ptr;  
        ptr = ptr->next;  
    }  
    if (prev == NULL)  
        return new;  
    prev->next = new;  
    return head;  
}
```

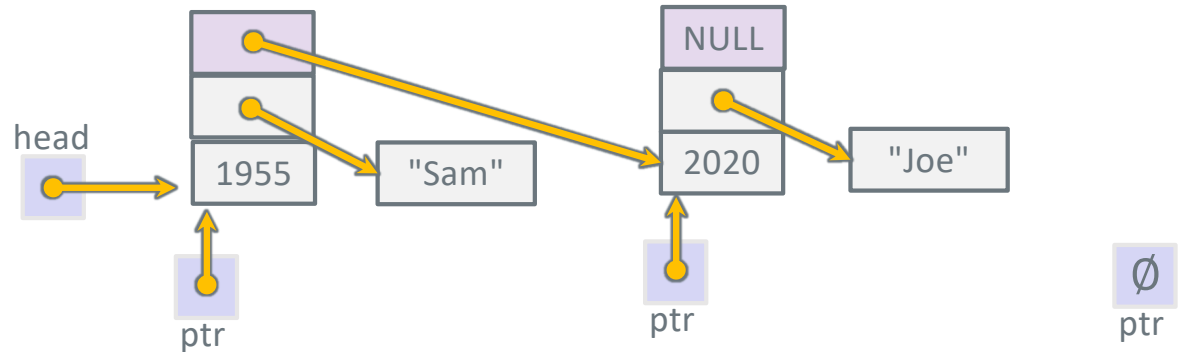


```
struct node *head = NULL; // insert at end  
struct node *ptr;  
if ((ptr = insertEnd(2020, "Joe", head)) != NULL)  
    head = ptr;  
if ((ptr = insertEnd(1955, "Sam", head)) != NULL)  
    head = ptr;
```

"Dumping" the Linked List

"walk the list from head to tail"

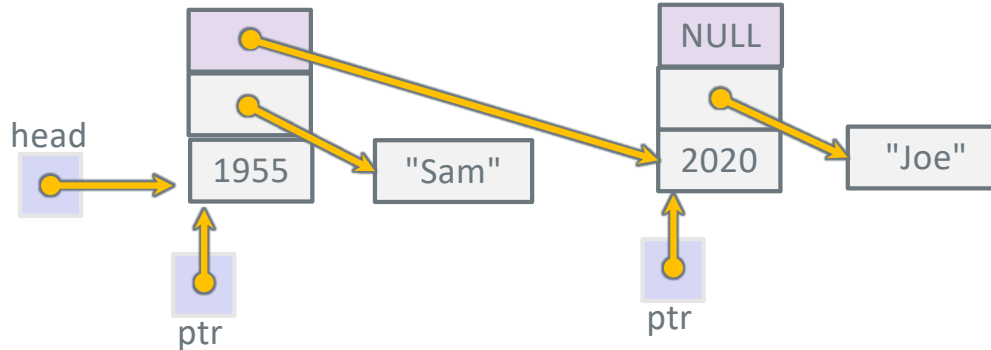
```
struct node {  
    int year;  
    char *name;  
    struct node *next;  
};
```



```
struct node *head;  
struct node *ptr;  
...  
printf("\nDumping All Data\n");  
ptr = head;  
while (ptr != NULL) {  
    printf("year: %d name: %s\n", ptr->year, ptr->name);  
    ptr = ptr->next;  
}
```

Dumping All Data
year: 1955 name: Sam
year: 2020 name: Joe

Finding A Node Containing a Specific Payload Value



```
struct node {
    int year;
    char *name;
    struct node *next;
};
```

```
struct node *findNode(char *name, struct node *ptr)
{
    while (ptr != NULL) {
        if (strcmp(name, ptr->name) == 0)
            break;
        ptr = ptr->next;
    }
    return ptr;
}
```

Returns pointer if found
NULL otherwise

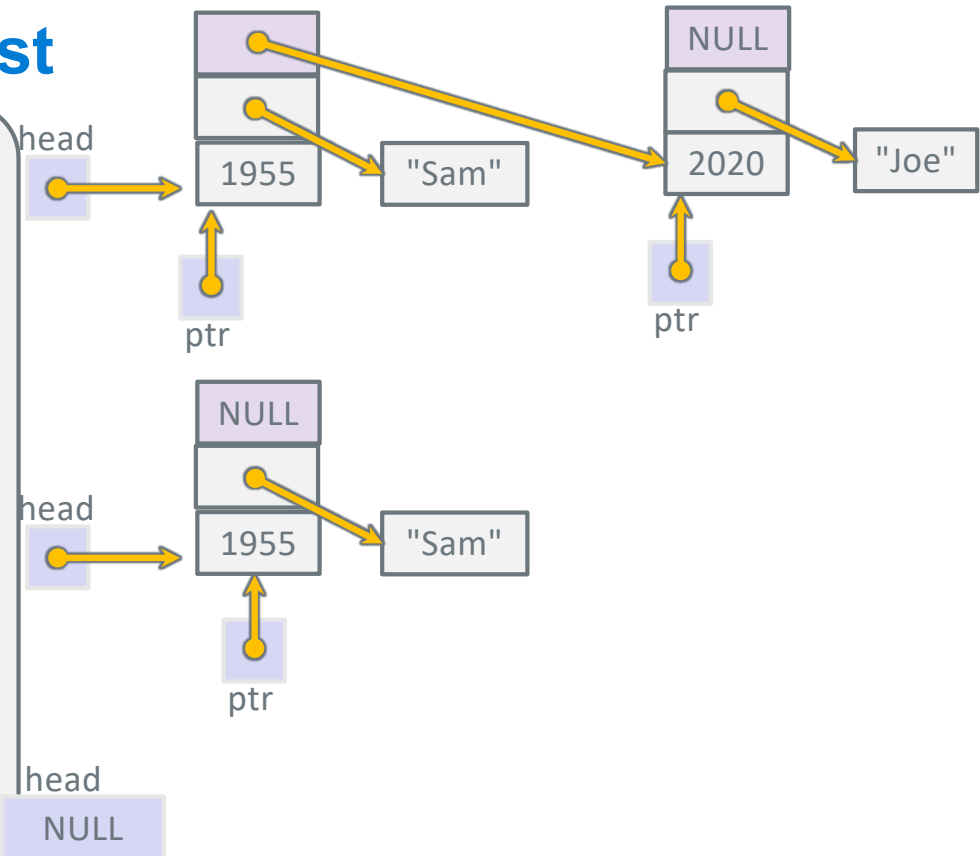
```
struct node *found;
```

```
if ((found = findNode("Joe", head)) != NULL)
    printf("year: %d name: %s\n", found->year, found->name);
```

Deleting a Node in a Linked List

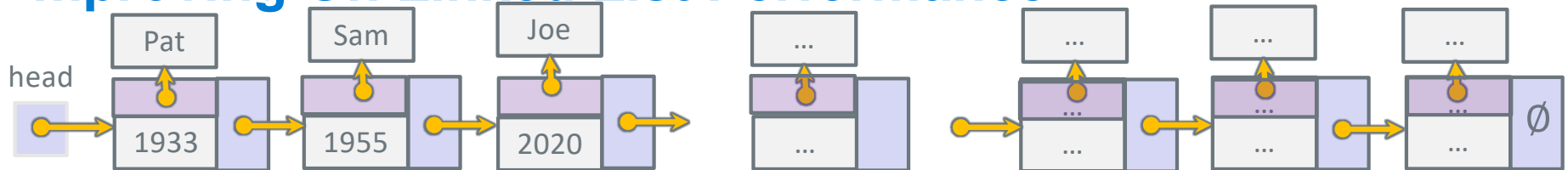
```
// returns head pointer; may have changed...
struct node *deleteNode(int name, struct node *head)
{
    struct node *ptr = head;
    struct node *prev = NULL;

    while (ptr != NULL) {
        if (strcmp(name, ptr->name) == 0)
            break;
        prev = ptr;
        ptr = ptr->next;
    }
    if (ptr == NULL) // not found return head
        return head;
    if (ptr == head) // remove first node new head
        head = ptr->next;
    else
        prev->next = ptr->next; // remove not head
    free(ptr->name); // free strdup() space
    free(ptr);
    return head;
}
```



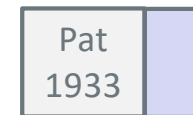
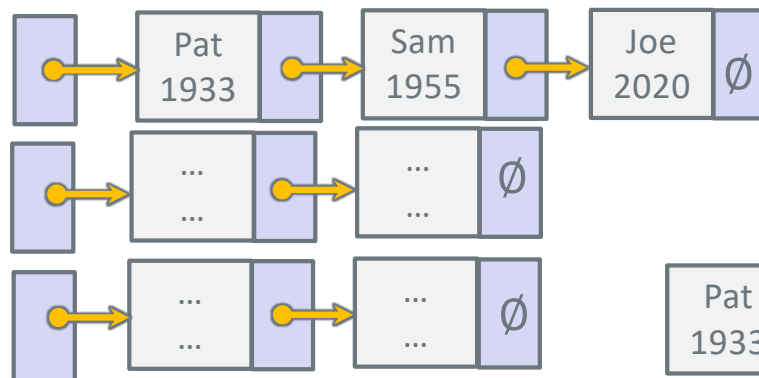
```
struct node *head = NULL;
head = deleteNode("Joe", head);
head = deleteNode("Sam", head);
```


Improving On Linked List Performance

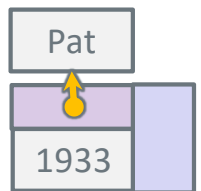


- When linked lists get long, the cost of finding an entry continues to increase $O(n)$
- How to improve search time?
- Break the single linked list into multiple **shorter length** linked lists
 - Shorter lists are faster to search
- **Requires a function** that takes a **lookup key** and selects just one of the shortened lists

How do you determine on which linked list an entry is stored?



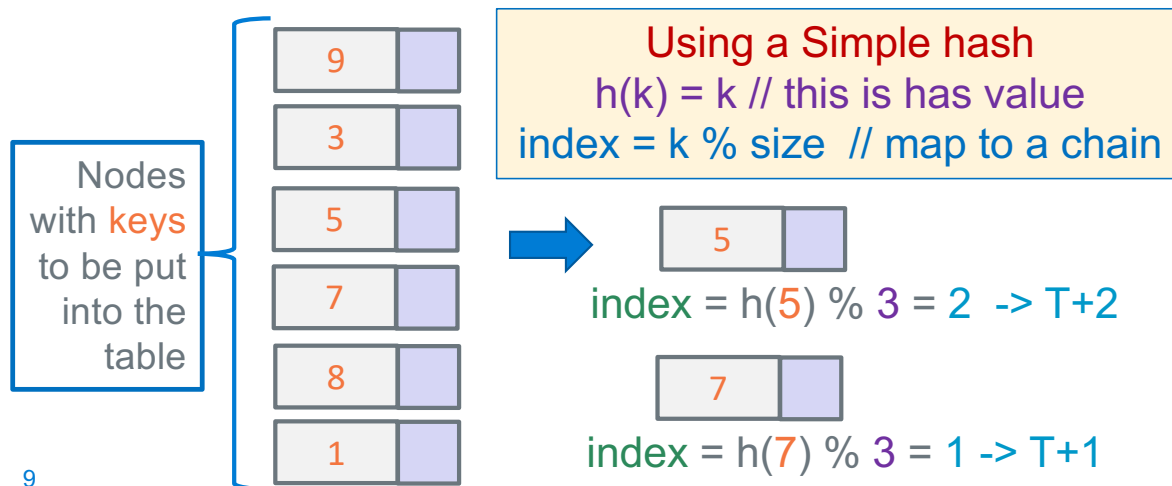
same as →



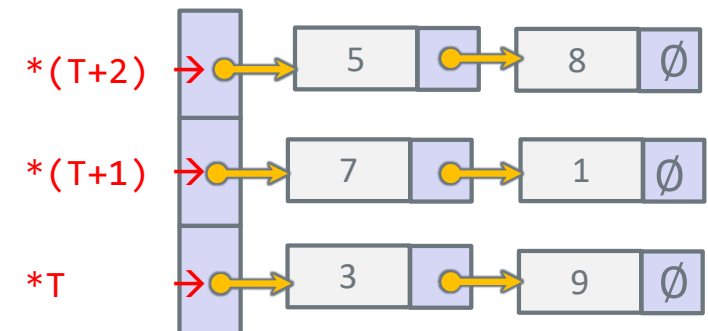
X

Hashing

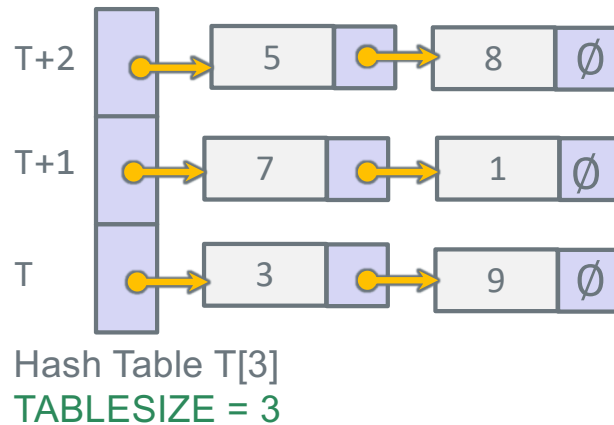
- Hash table is an array of **pointers** to the head of different linked lists (called hash chains)
- **Each data item** must have a **unique key** that identifies it (e.g., auto license plate)
 - $h(k)$ is the **hash value** of key k to *encode the key k into an integer*
- Use the Hash value to map to **one entry** in the hash table $T[]$ of size TABLESIZE
 - $\text{Index} = h(k) \% \text{TABLESIZE}$ (mod operator $\%$ maps a **keys** hash value to **table index**)
- **Keys** that hash to the same array index (*collide*) are put on a linked list
- After hashing a **key**, you then traverse the selected linked list to find the entry



Hash Table $T[3]$ of **linked list head pointers**
 TABLESIZE = 3



Hash Table With Collision Chaining (multiple linked lists)



- Make $TABLESIZE$ **prime** as keys are typically not randomly distributed, and have a *pattern*
 - Mostly even, mostly multiples of 10, etc.
 - In general: mostly multiples of some k
- If k is a factor of $TABLESIZE$, then only $(TABLESIZE/k)$ slots will ever be used!

1. Calculate index $i = \text{hash}(\text{key}) \% TABLESIZE$
 2. Go to array element i , i.e., $T+i$ that contains the head pointer for collision chain
 3. Walk the linked list for element, add element, remove element, etc. from the linked list
 4. New items added to the hash table are typically added at the front or at the end of the collision chain linked list (when multiple keys hash to same index .. they collide)
- Hash arrays need an index number to select a chain, so if we have a string, we must first convert to a number

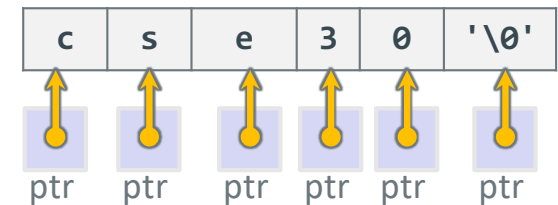
Simple 32-bit String Hash Function in C (djb2)

```
uint32_t hash(char *str)
{
    uint32_t hash = 0U;
    uint32_t c;

    while ((c = (unsigned char)*str++) != '\0')
        hash = c + (hash << 6) + (hash << 16) - hash;

    return hash;
}
```

Signed Data types	Unsigned Data types	Exact Size
int32_t	uint32_t	32 bits (4 bytes)



- Many different algorithms for string hash function (Dan Berman's djb2 above)
 - << is the **left** bit shift operator (later in course)
- **Fast to compute**, has a reasonable key distribution for **short 8-bit ASCII strings** into **32-bit unsigned ints**

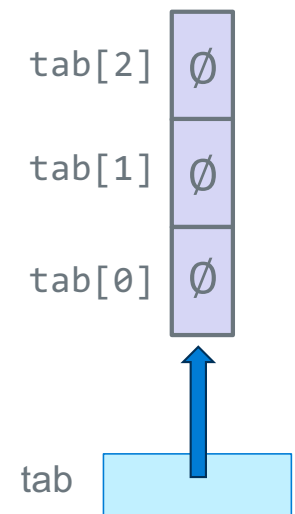
Allocating the Hash Table (collision chain head pointers)

Good use for calloc()

```
#define TBSZ 3
int main(void)
{
    struct node *ptr;
    struct node **tab; // pointer to hashtable
    uint32_t index;

    if ((tab = calloc(TBSZ, sizeof(*tab))) == NULL) {
        fprintf(stderr, "Cannot allocate hash
table\n");
        return EXIT_FAILURE;
    }
    // continued on next slide
```

TABLESIZE = 3



Inserting Nodes into the Hash Table (at the end)

```
#define TBSZ 3
unit32_t index;

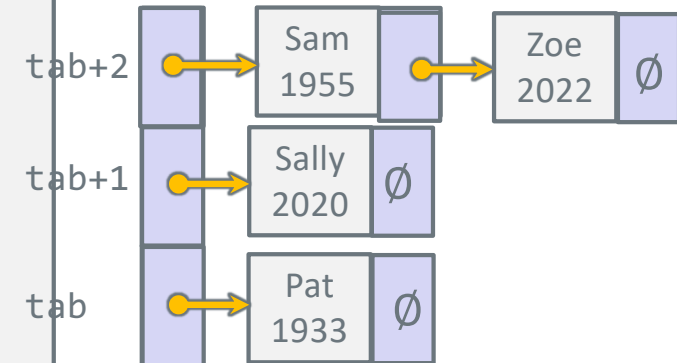
index = hash("Pat") % TBSZ;
if ((ptr = insertEnd(1933, "Pat", *(tab + index))) != NULL)
    *(tab + index) = ptr;

index = hash("Sam") % TBSZ;
if ((ptr = insertEnd(1955, "Sam", *(tab + index))) != NULL)
    *(tab + index) = ptr;

index = hash("Sally") % TBSZ;
if ((ptr = insertEnd(2020, "Sally", *(tab + index))) != NULL)
    *(tab + index) = ptr;

index = hash("Zoe") % TBSZ;
if ((ptr = insertEnd(2022, "Zoe", *(tab + index))) != NULL)
    *(tab + index) = ptr;
```

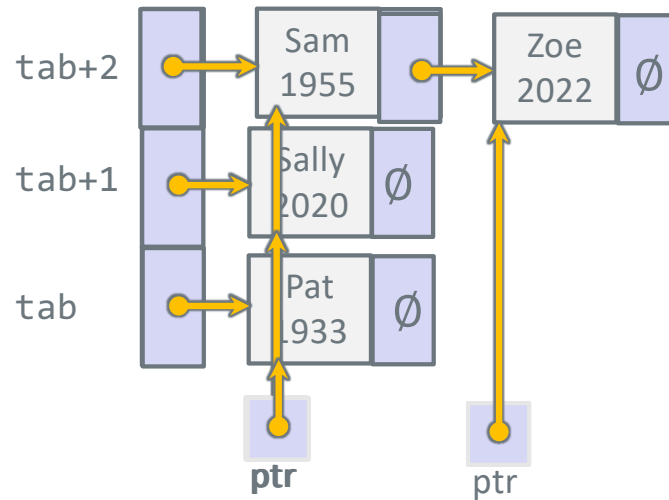
```
struct node {
    int year;
    char *name;
    struct node *next;
};
```



Notice

Substitute **createNode()** for **insertEnd()** to insert nodes at the **front** of the collision chain **instead** of at the **end** of the collision chain

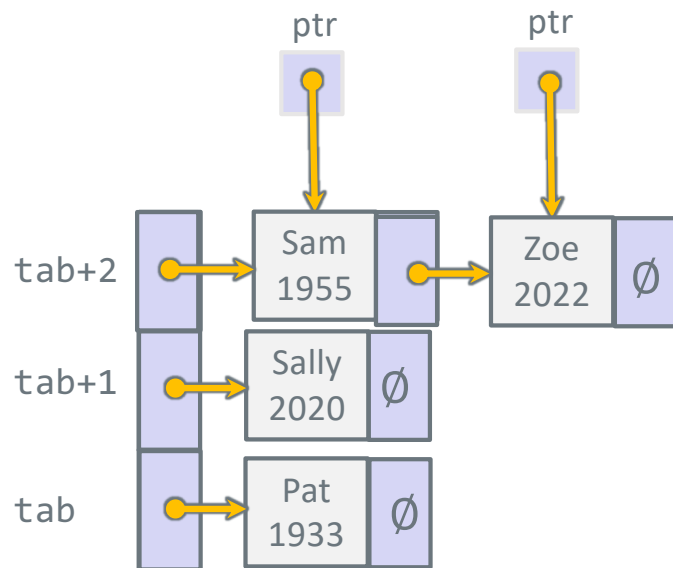
"Dumping" the Hash Table (traversing all Nodes)



Dumping All Data
chain: 0
Year: 1933 Name: Pat
chain: 1
Year: 2020 Name: Sally
chain: 2
Year: 1955 Name: Sam
Year: 2022 Name: Zoe

```
printf("\nDumping All Data\n");  
for (index = 0U; index < TBSZ; index++) {  
    ptr = *(tab + index);  
    printf("chain: %d\n", index);  
  
    while (ptr != NULL) {  
        printf("Year: %d Name: %s\n", ptr->year, ptr->name);  
        ptr = ptr->next;  
    }  
}
```

Finding a Node with a Specific Payload Value



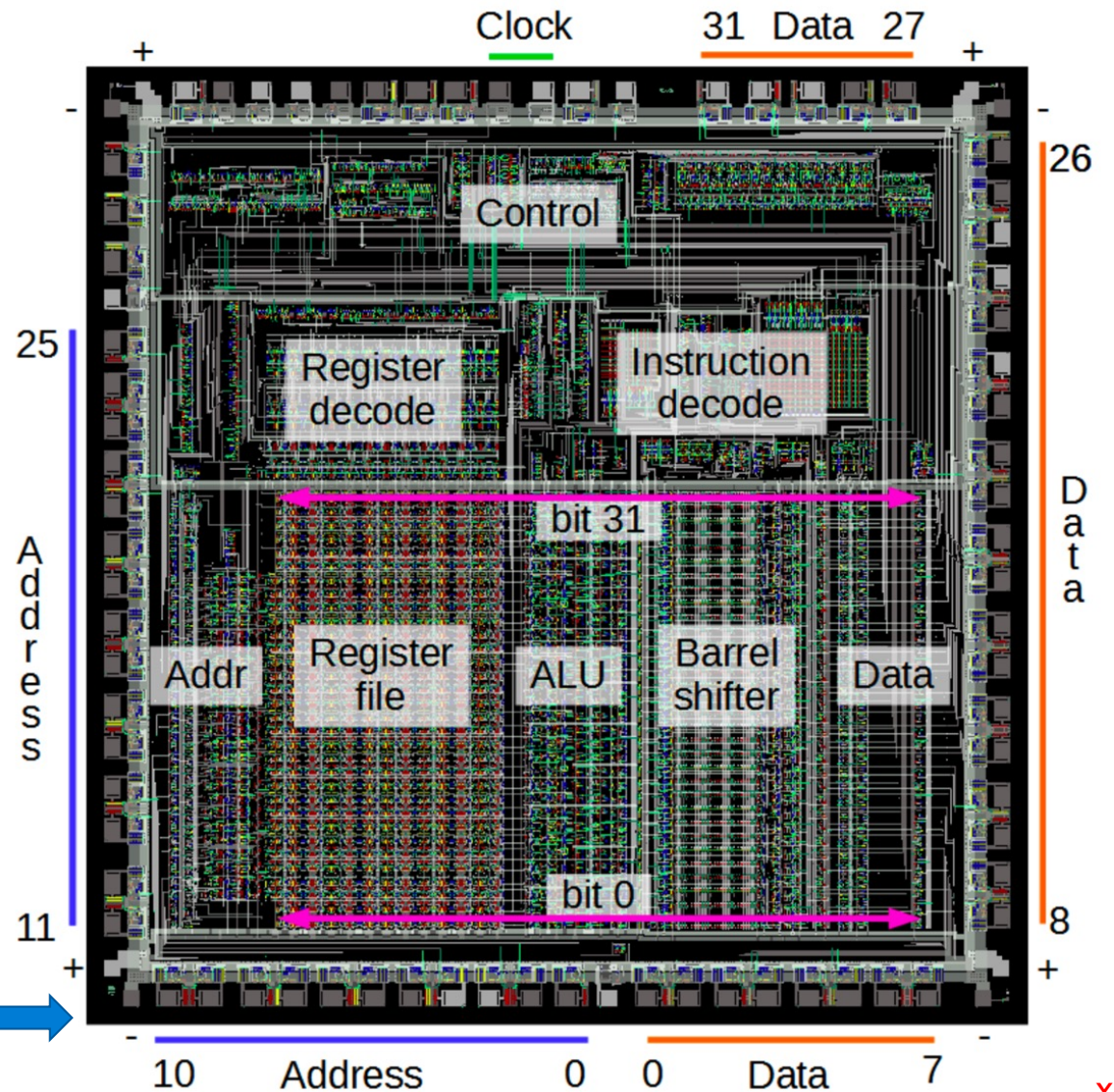
```
// same routine as shown in a previous slide
struct node *findNode(char *name, struct node *ptr)
{
    while (ptr != NULL) {
        if (strcmp(name, ptr->name) == 0)
            break;
        ptr = ptr->next;
    }
    return ptr;
}
```

```
index = hash("Zoe") % TBSZ;
if ((ptr = findNode("Zoe", *(tab + index))) != NULL)
    printf("Found Year: %d name: %s\n", ptr->year, ptr->name);
else
    printf("Not Found Zoe\n");
```

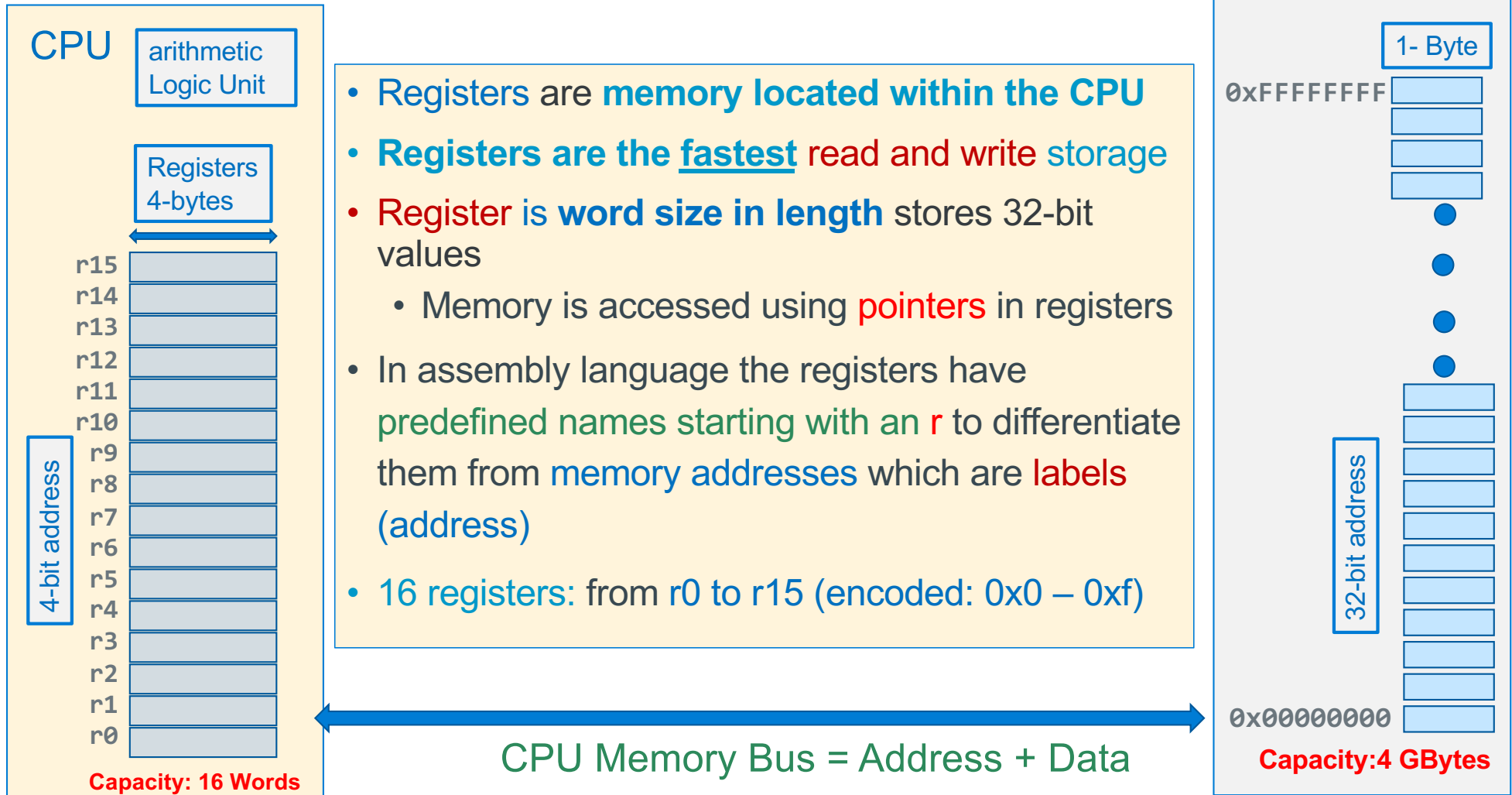

Arm Core Floorplan

- **Control:** Specifies the operation of the CPU
- **Register File:** Memory inside the CPU
 - Instructions reference these directly
- **ALU:** Arithmetic Logic Unit: Arithmetic and bitwise hardware (on the bits)
- **Barrel shifter:** (shifts bits in a register during instruction execution - Later)
- **Instruction Decode:** Interprets the the bits in an instruction to determine what the instruction means
- **Register Decode:** controls the registers in during instruction execution
- **Address and Data:** Interface to external RAM (like memory dimms)

Single core *arm* die *Floorplan*



32-Bit Arm - Registers



Using ARM-32 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**

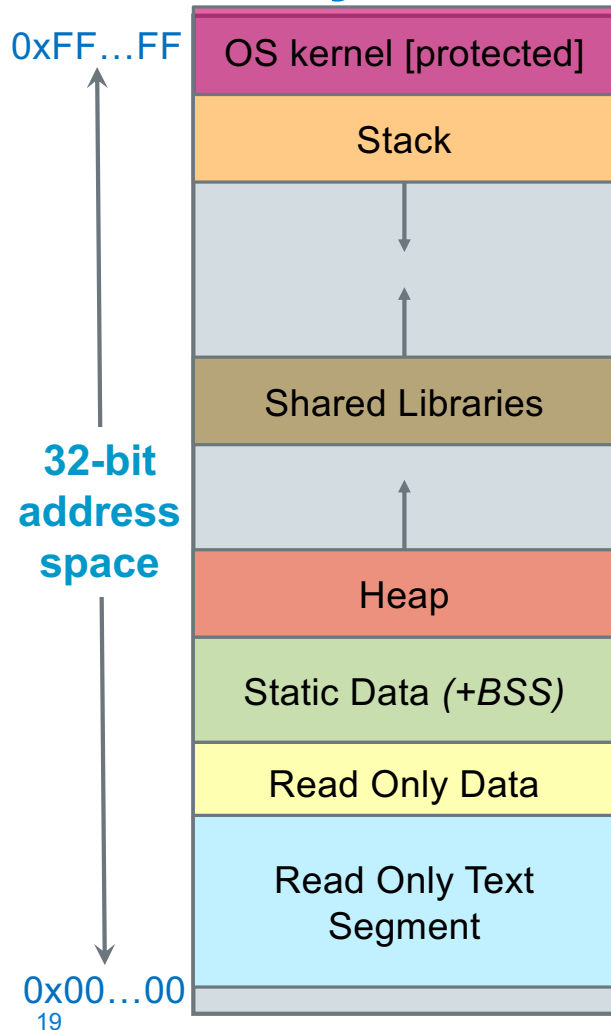
r3

r2

r1

r0

Assembly and Machine Code



- Machine Language (or code): Set of instructions the CPU executes are encoded in memory using patterns of ones and zeros (like binary numbers)
- Assembly language is a **symbolic version** of the machine language
- Each assembly statement (called an **Instruction**)
 - Executes **exactly one** from a list of simple commands
 - Instructions describe operations (e.g., =, +, -, *)
 - Execution proceeds from low to high memory one instruction at a time unless there is a branch
- One line of **arm32 machine code** contains one instruction in one word (32 bits)

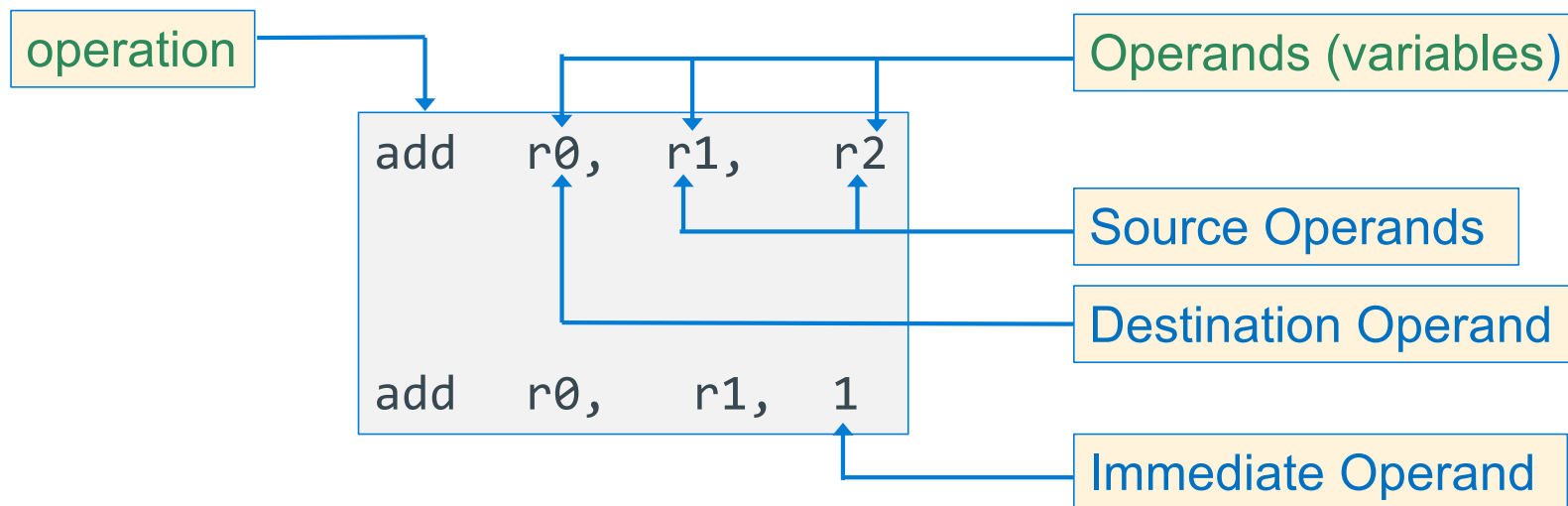
Instruction Address	Instruction (4-bytes) contents	Assembly Equivalent
1040c:	e28db004	add fp, sp, 4
10410:	e59f0010	ldr r0, [pc, 16]
10414:	ebffffb3	bl 102e8 <printf>
10418:	e3a00000	mov r0, 0
1041c:	e24bd004	sub sp, fp, 4

Machine Code

high <- low bytes

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

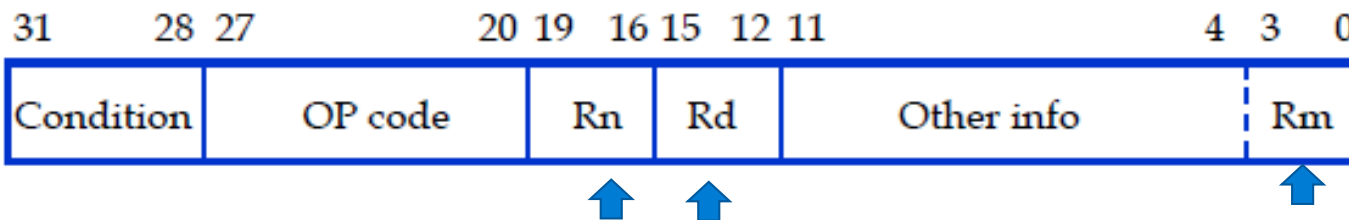
4-bytes

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

- 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,...

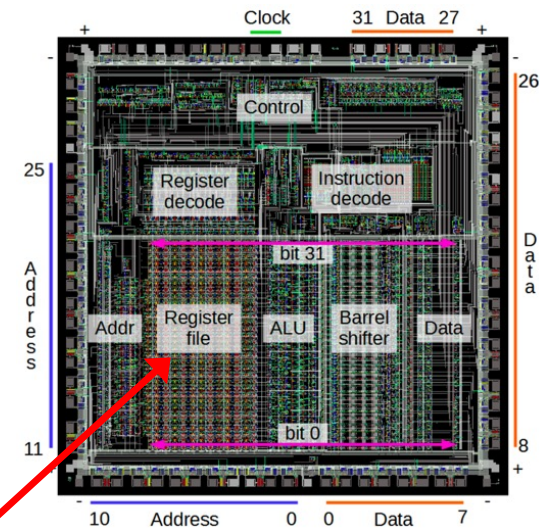
Memory Content in Text segment

add r0, r1, r1 Low memory

add r0, r0, r0

add r0, r0, r0

sub r1, r0, r1 High memory



Register contents inside the CPU

r0 = 1 r1 = 2 initial values

r0 = 4 r1 = 2

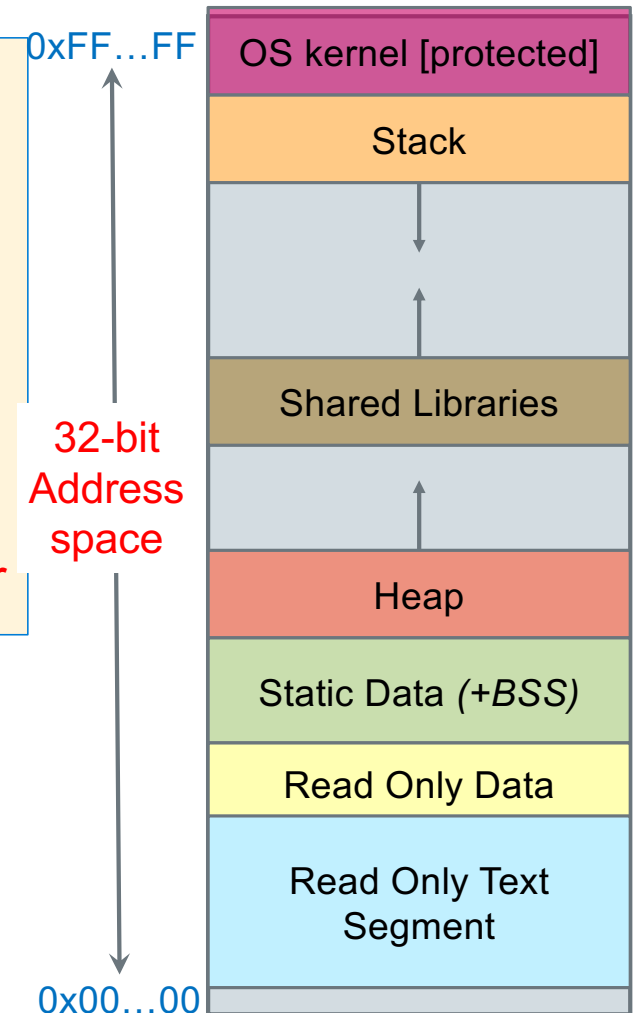
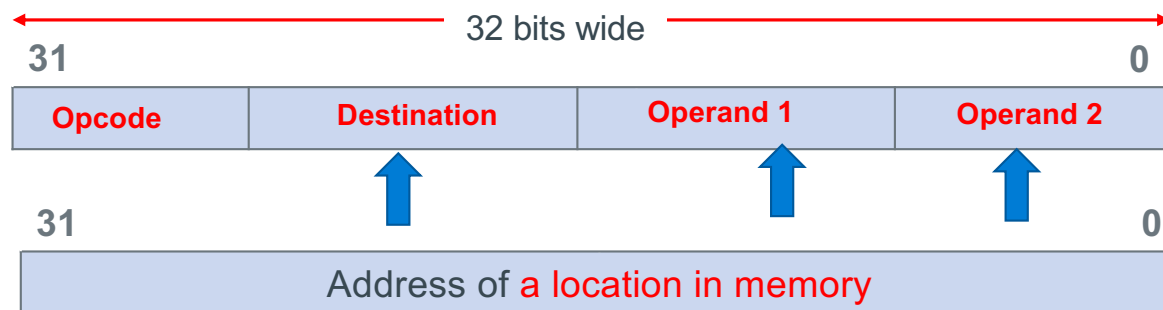
r0 = 8 r1 = 2

r0 = 16 r1 = 2

r0 = 16 r1 = 14

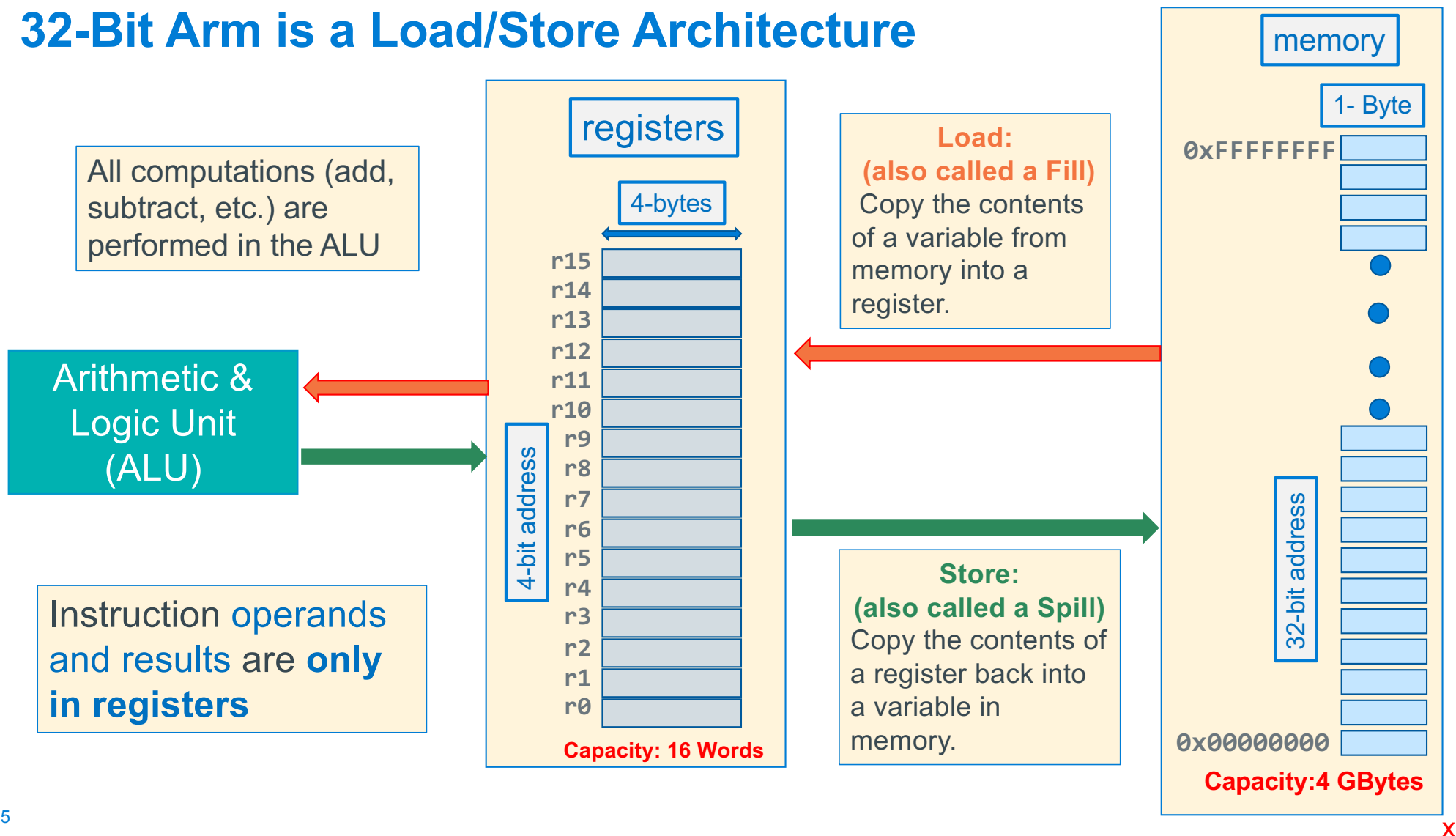
How to Access Memory?

- 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

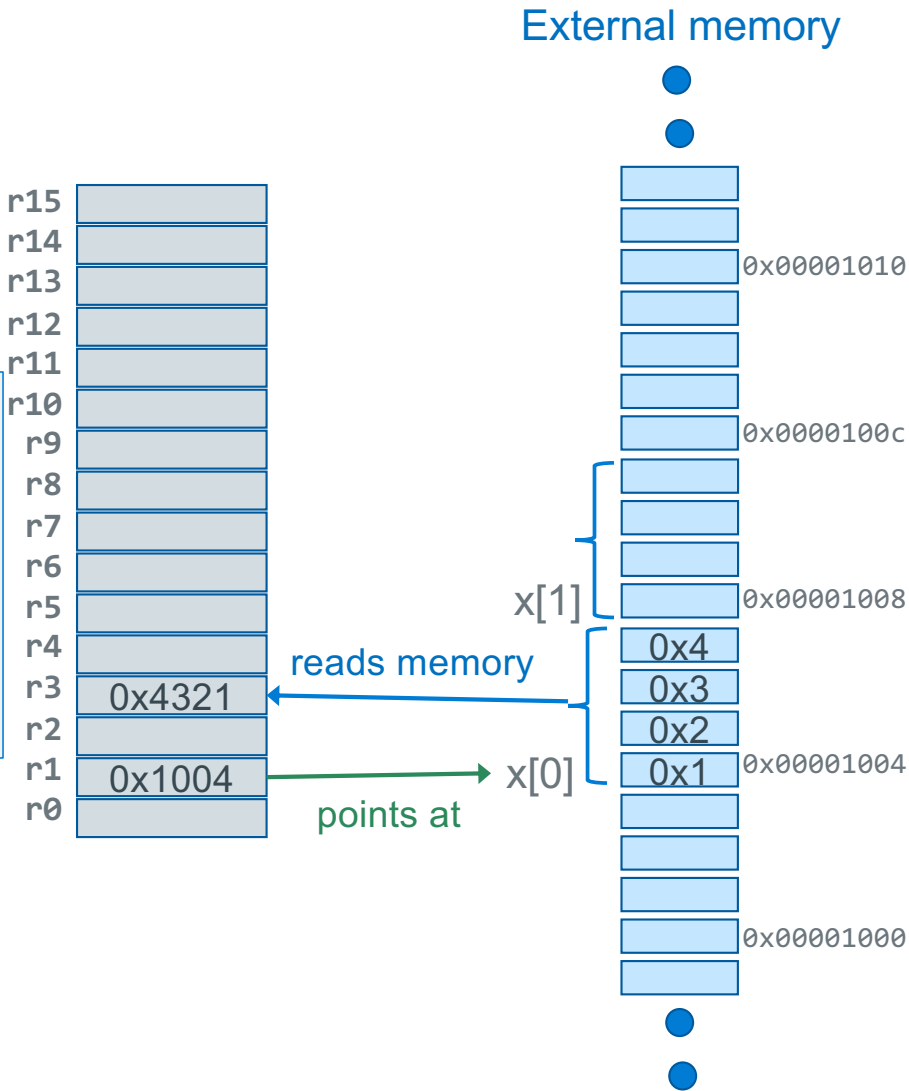
32-Bit Arm is a Load/Store Architecture



Load/Store: Load

```
int x[2] = {0x4321, 0x0};  
x[1] = x[0];
```

```
// load store concept  
  
int r3;  
int x[2];  
int *r1;  
  
r1 = &x;           // r1 contains x's address  
r3 = *(r1);         // read memory , load register 3
```



Load/Store: Store

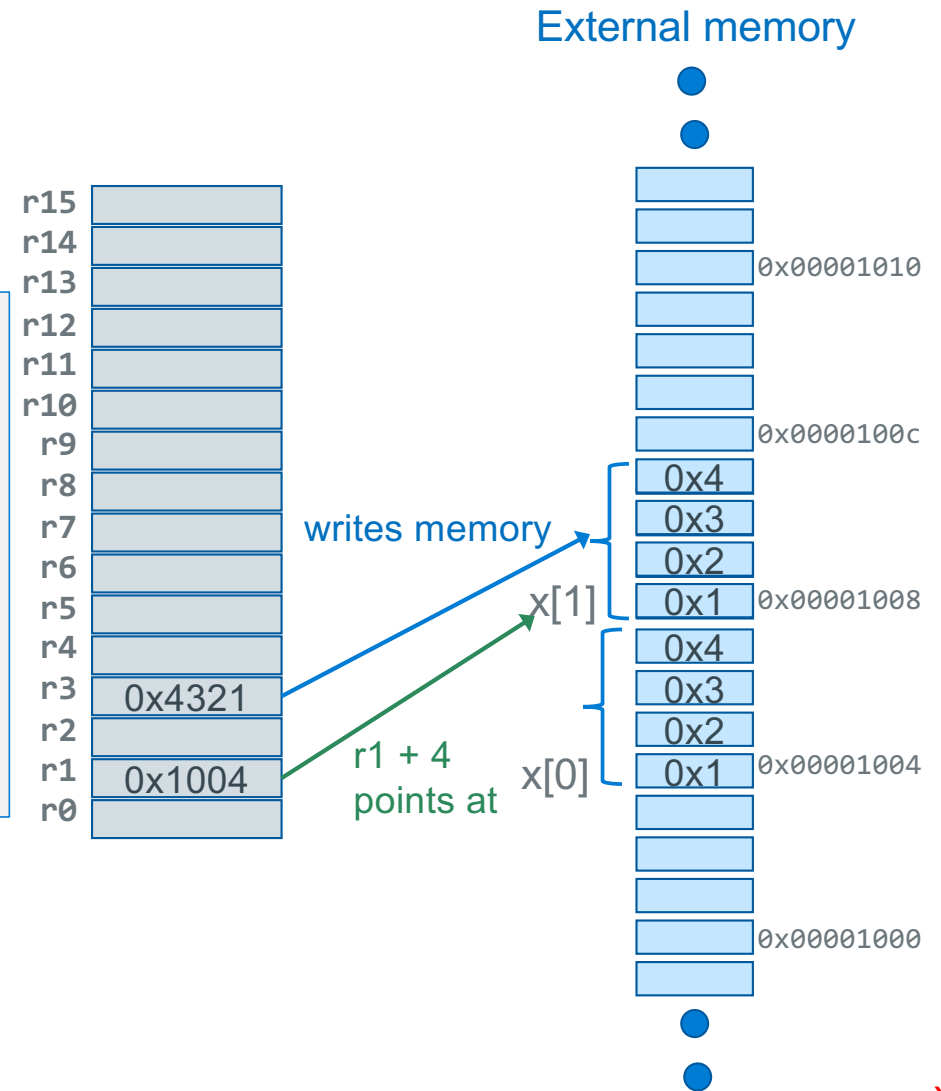
```
int x[2] = {0x4321, 0x0};  
x[1] = x[0];
```

```
// load store concept
```

```
int r3;  
int x[2];  
int *r1;
```

```
r1 = &x;           // r1 contains x's address  
r3 = *(r1);        // read memory , load register 3
```

```
r1 = r1 + 1;  
*(r1) = r3;        // store register 3 to memory
```



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

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 Between Registers - MOV

```
mov  r0, r1
```

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

register direct "addressing"

register r1



register r0

```
mov  r0, 100
```

```
// Expands an 8-bit (imm8)  
value 100  
// stored in the instruction  
// into the register r0
```

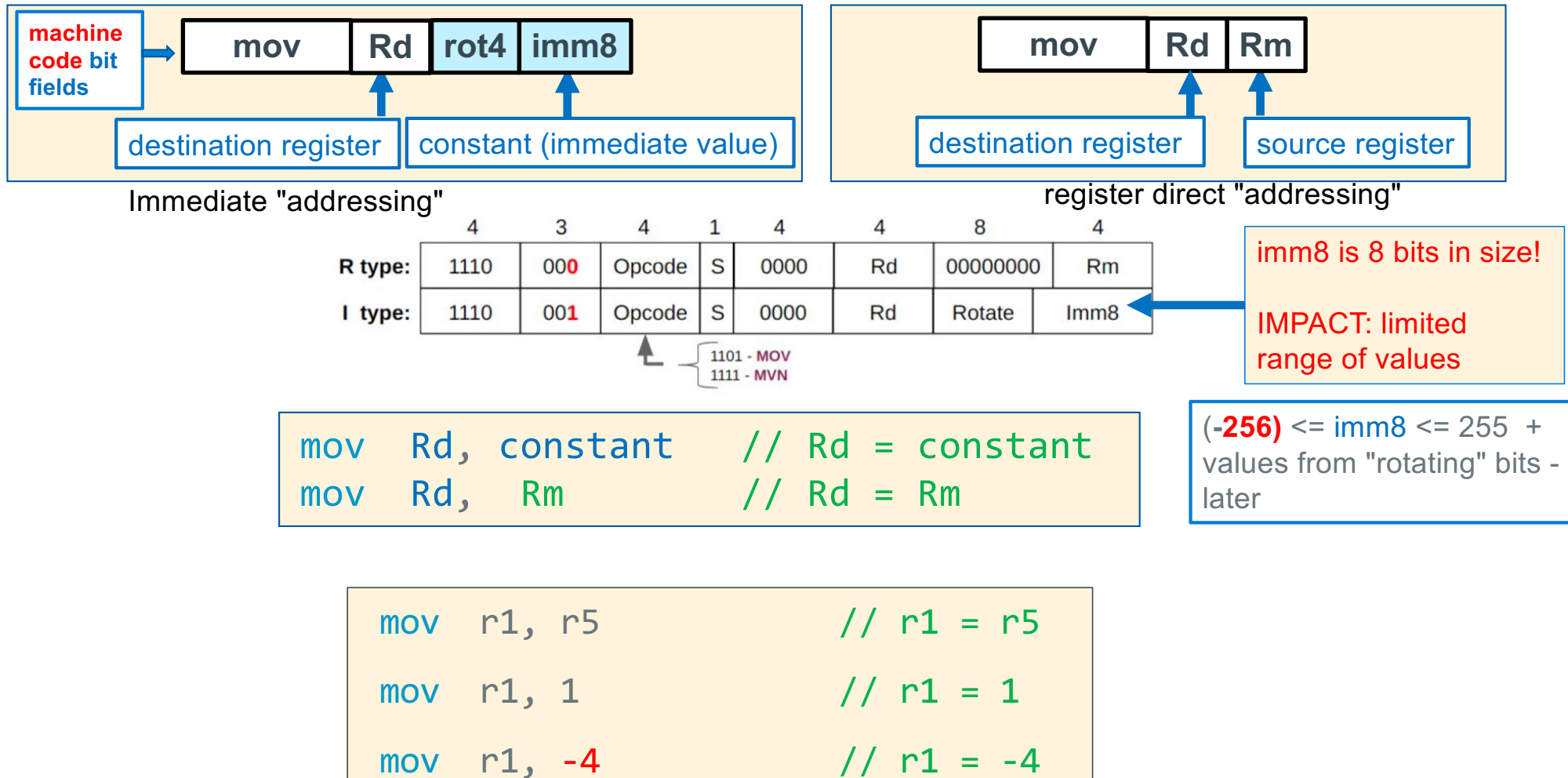
Immediate "addressing"

100

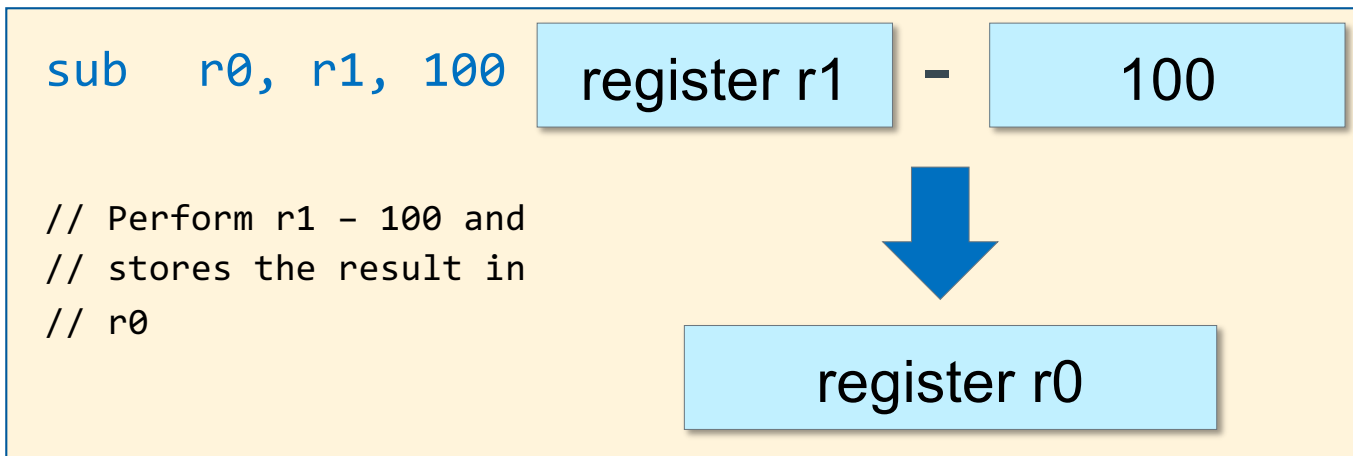
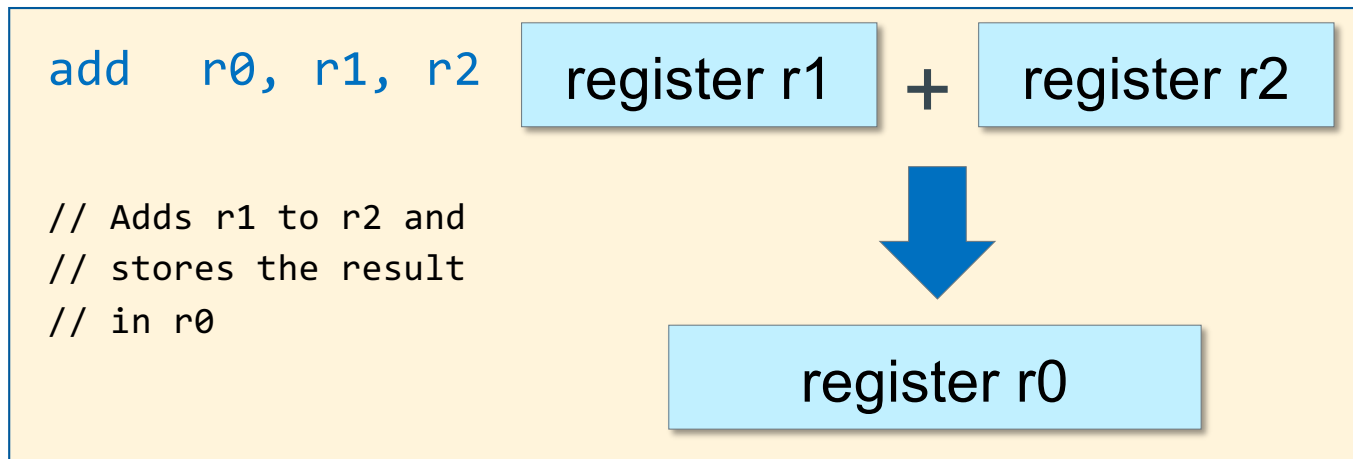


register r0

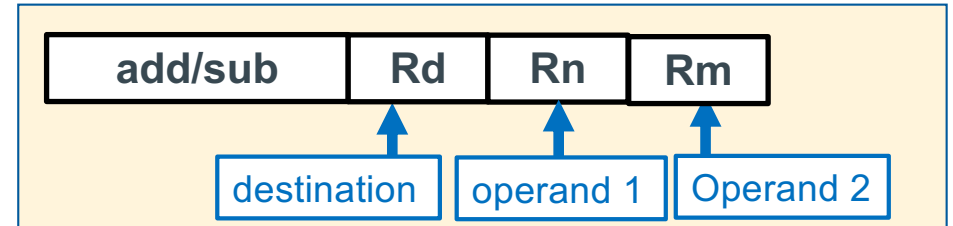
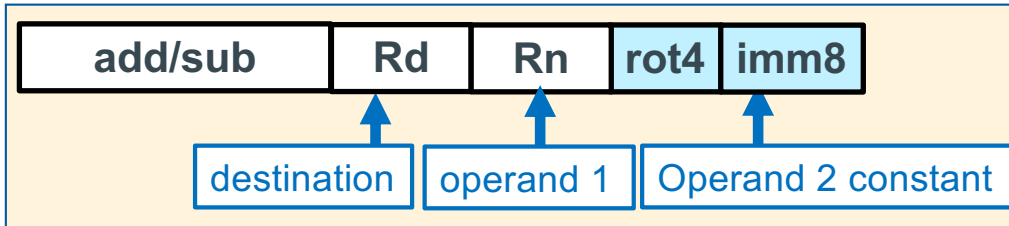
mov – Copies Register Content between registers



First Look: Add/Sub Registers



add/sub – Add or Subtract two integers



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

```
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
```

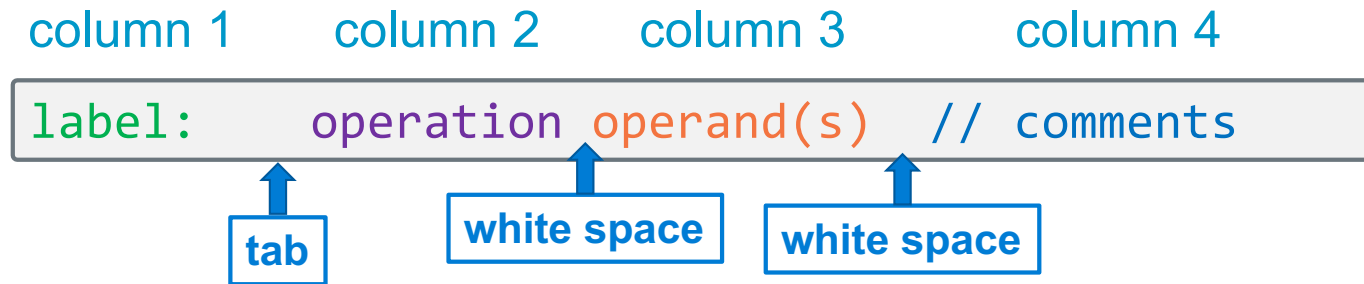
```
a = b + c + d - e;
r0 = r1 + r2 + r3 - r4;
r0 = ((r1 + r2) + r3) - r4;
r0 = r1 + r2;
r0 = r0 + r3
r0 = r0 - r4
```

```
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
```

Line Layout in an Arm Assembly Source



- 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**

Labels in Arm Assembly - 1

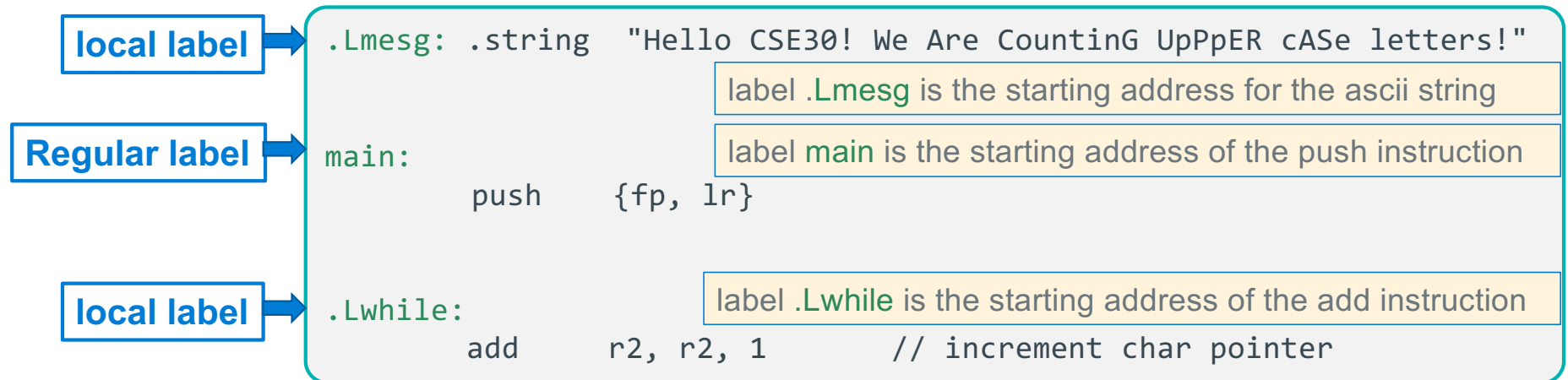
```
label:  operation operand(s)  // comment

        // 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 (often on a line by itself ABOVE the "operation")
 - **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 **line's** 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 - 2



- 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 **literal** not the address of **foo** in the following)
`char *foo = "literal";`

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



imm24 is Relative direction
from the branch instruction (in +/- instructions)

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

b **.Llabel**

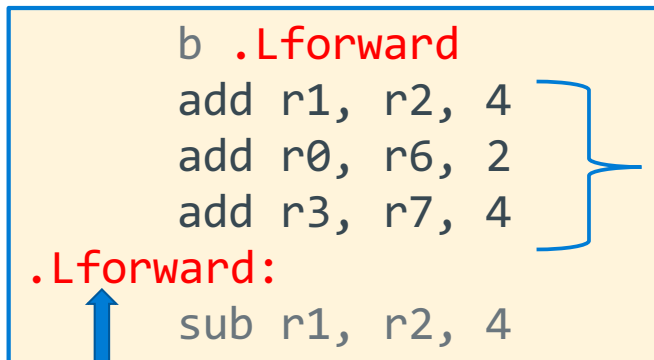
- Causes an unconditional branch (aka goto) to the instruction with the address **.Llabel**
- **.Llabel** is called a **branch target label** (the "*target*" of a branch instruction)
- **Be careful! do not to branch to a function label!**
- **.Llabel**: translates into an number offset being **imm24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

```
        b        .Ldone                // goto .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
```



Not a
practical
example as
this code is
unreachable

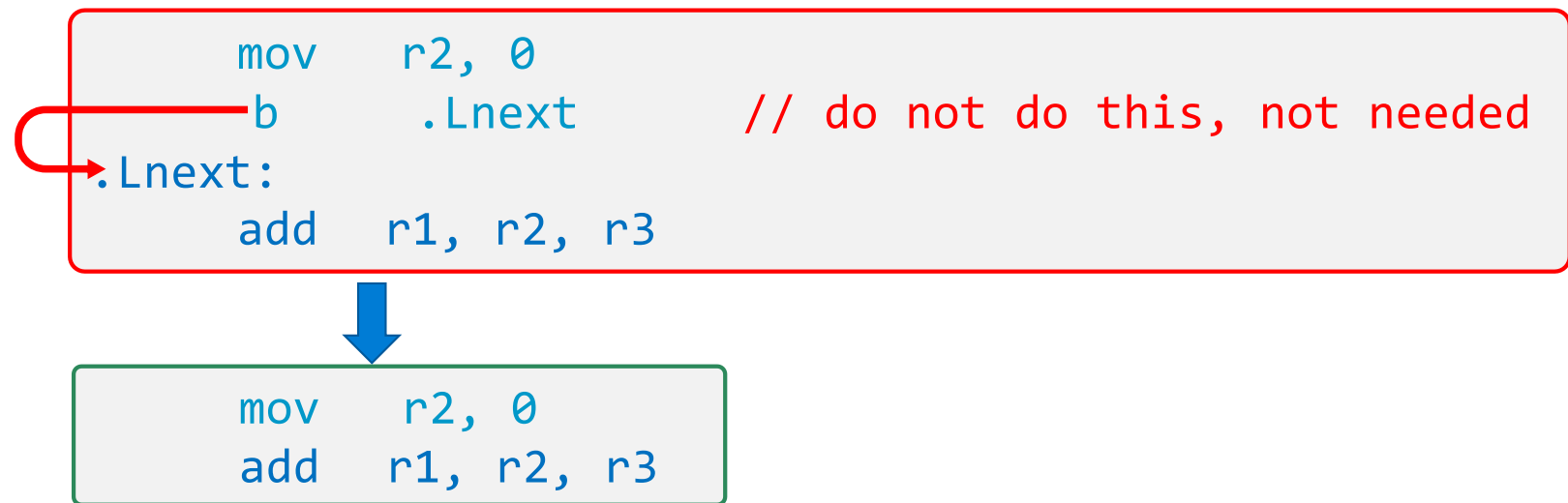
Branch target (local label)

Backward Branch (Infinite loop)

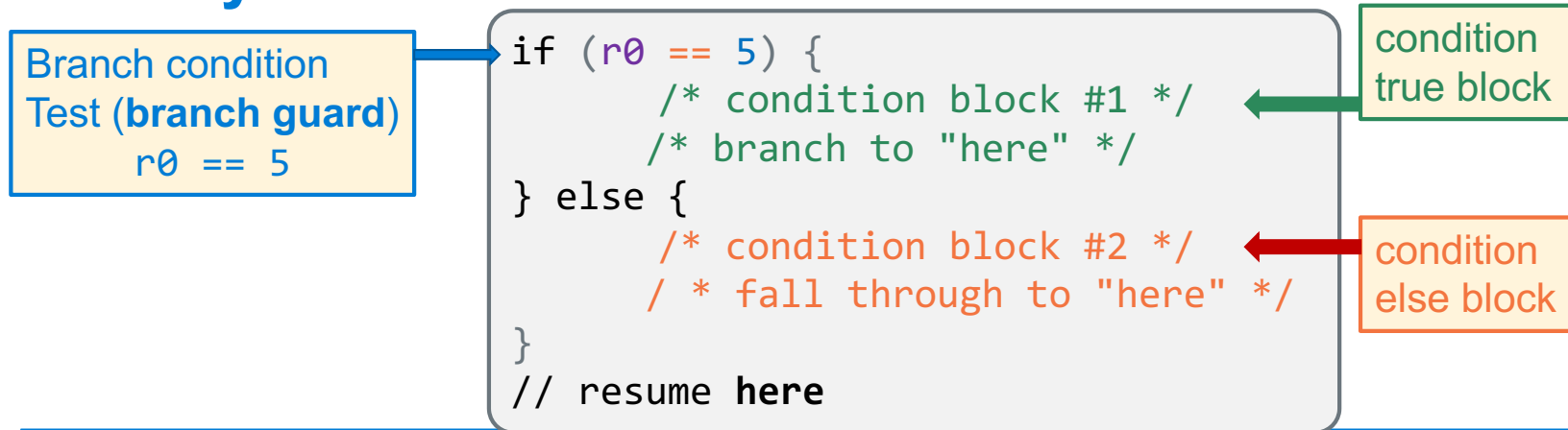
```
.Lbackward:
    add r1, r2, 4
    sub r1, r2, 4
    add r4, r6, r7
    b .Lbackward
// not reachable unless
// there is a label after the .b above
```

- 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!

Never Branch to the following instruction: It is not needed!

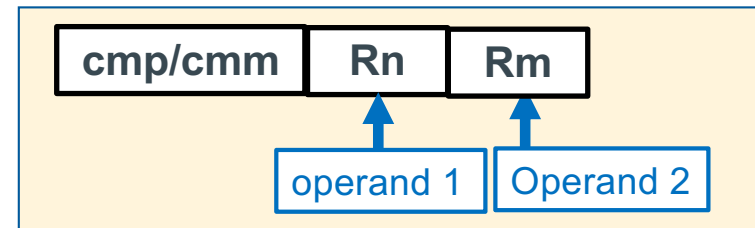
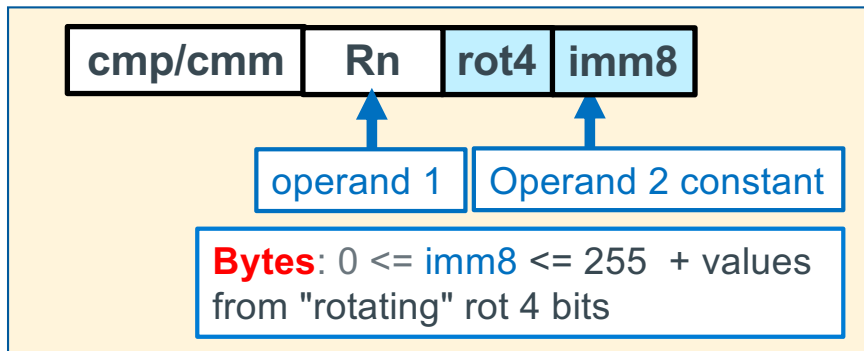


Anatomy of a Conditional Branch: If statement



- **Branch guard**: determines when to execute the "condition true block" or the "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 (else)** block
- Step 1: evaluate the branch guard(s) (involves one or more compares/tests)
- Step 2: If branch guard evaluates to be **false**
 - **branch around** the **true block** and execute the **else block**
 - **otherwise "fall through"** and execute the **true block**
- **Block order** in C is where the **True Block** appears above the **False 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
    
```