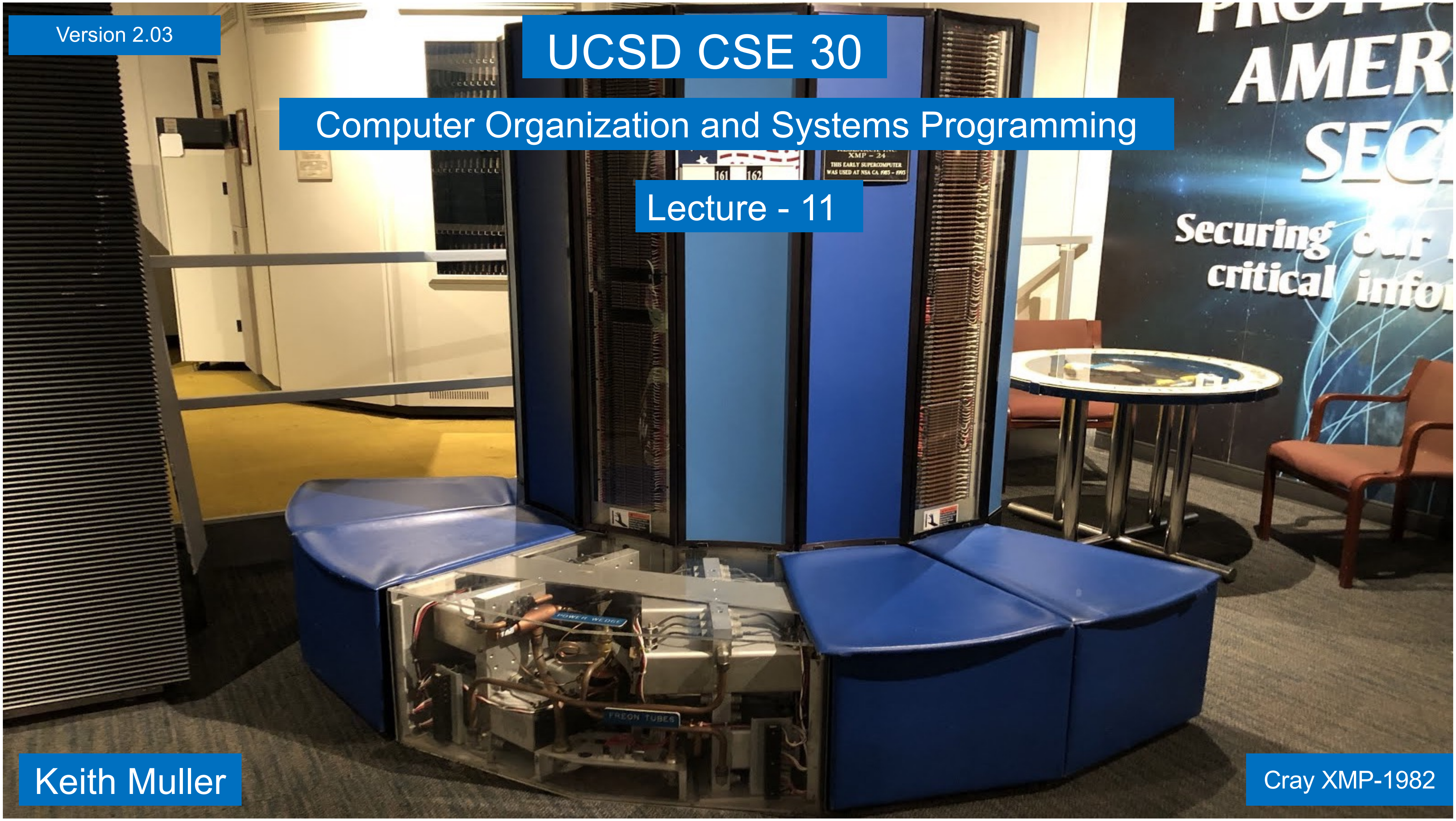
Version 2.03

UCSD CSE 30

Computer Organization and Systems Programming

Lecture - 11



Keith Muller

Cray XMP-1982

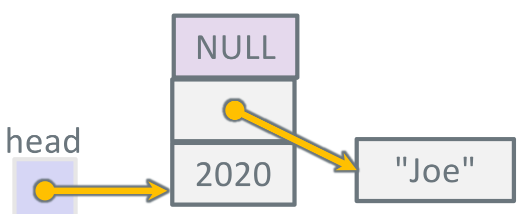
Creating a Node & Inserting it at the Front of the List

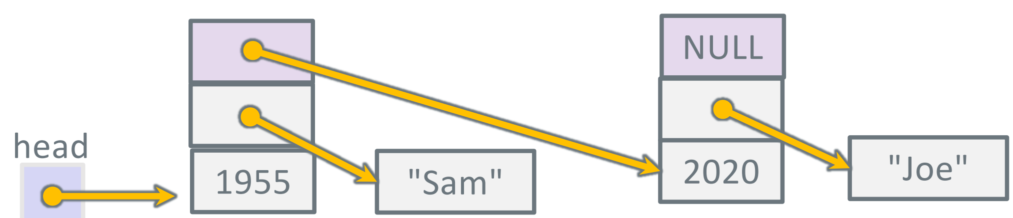
```
// create node; inserts at front when passed head
struct node *
creatNode(int year, char *name, struct node *link)
{
    struct node *ptr = malloc(sizeof(*ptr));
    if (ptr != NULL) {
        if ((ptr->name = strdup(name)) == NULL) {
            free(ptr);
            return NULL;
        }
   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

```
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
}
```

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

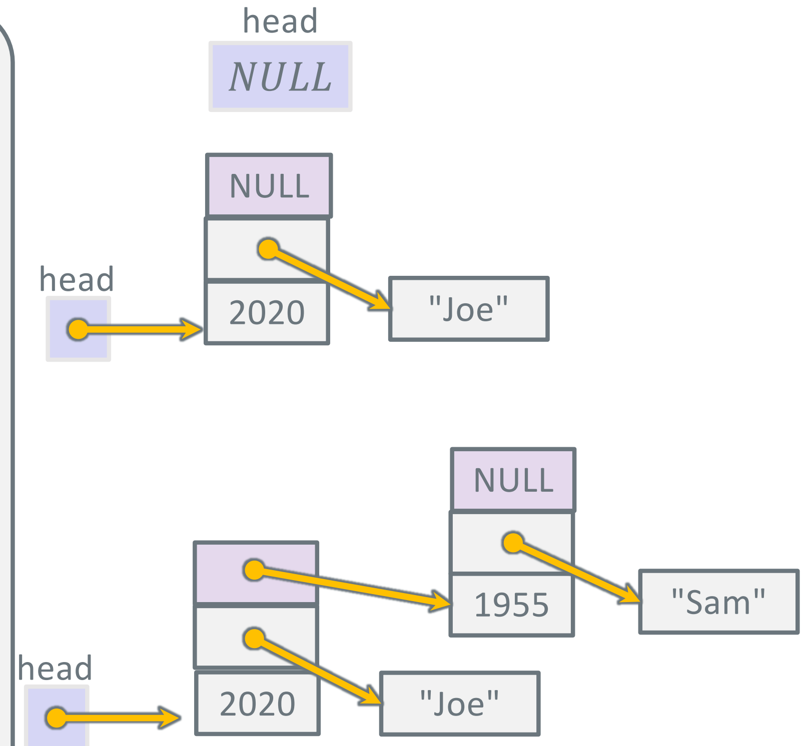




x

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;
}
```



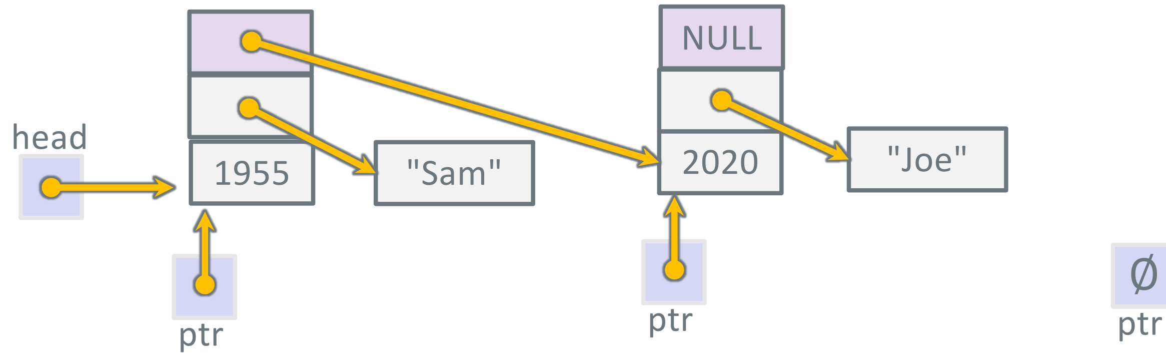

```
struct node *head = NULL;  // insert at end
struct node *ptr;
if ((ptr = insertEnd(2020, "Joe", head)) != NULL)
    head = ptr;
if ((ptr = insertEnd(1955, "Sam", head)) != NULL)
    head = ptr;
```


x

"Dumping" the Linked List

"walk the list from head to tail"

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

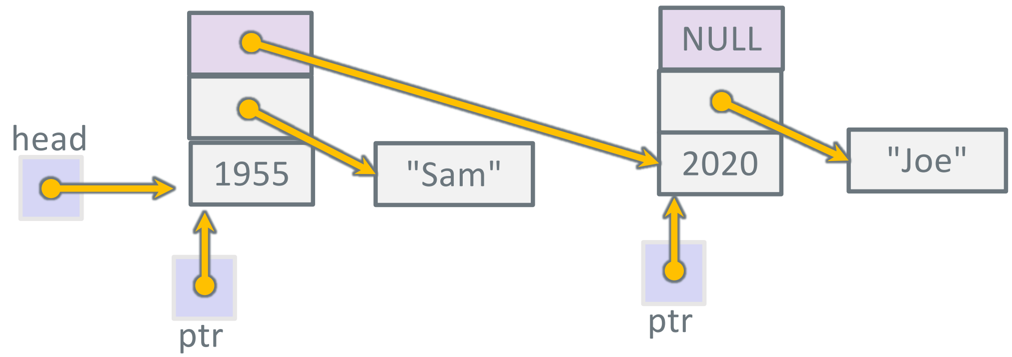



```
struct node *head;
struct node *ptr;
…
printf("\nDumping All Data\n");
ptr = head;
while (ptr != NULL) {
    printf("year: %d name: %s\n", ptr->year, ptr->name);
    ptr = ptr->next;
}
```

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


x

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;
}
```

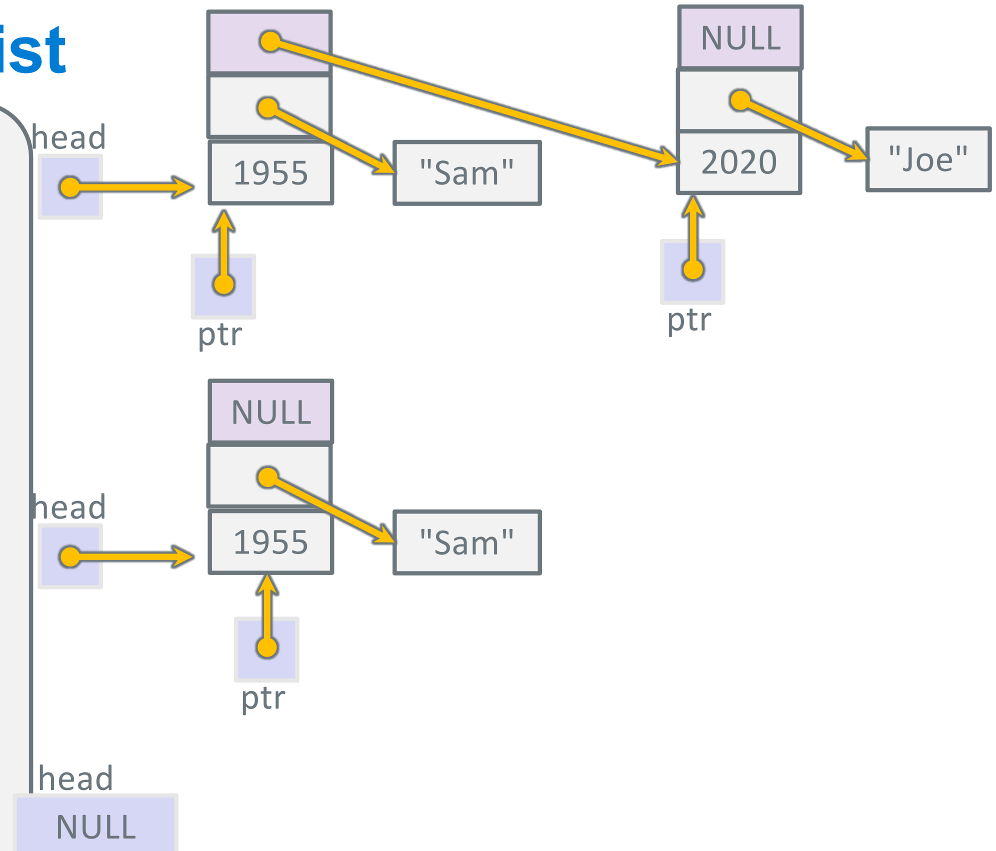
Returns pointer if found
NULL otherwise

```
struct node *found;

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


x

Deleting a Node in a Linked List

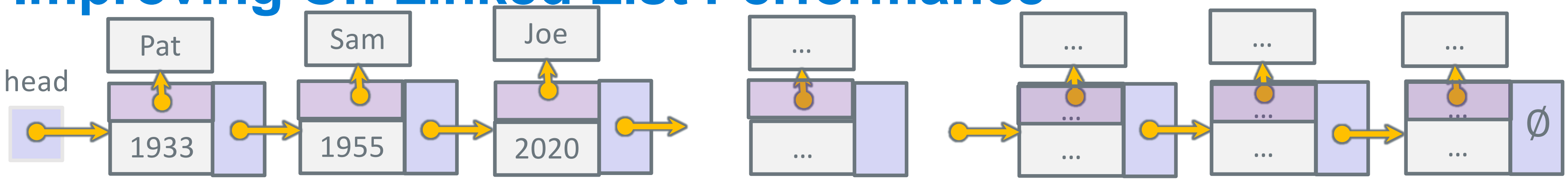
```
// returns head pointer; may have changed…
struct node *deleteNode(int name, struct node *head)
{
    struct node *ptr = head;
    struct node *prev = NULL; // base case

    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);
```

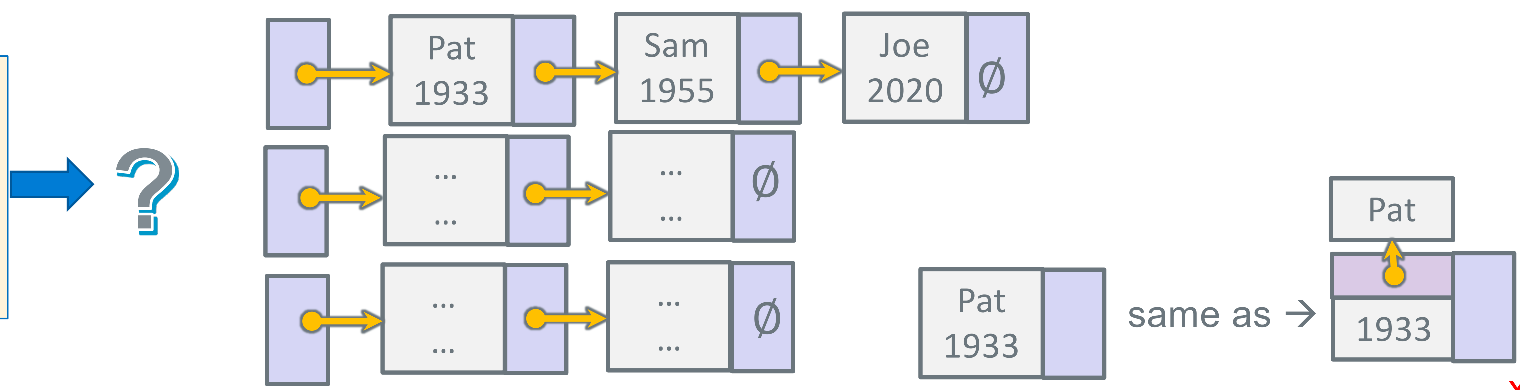

x

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

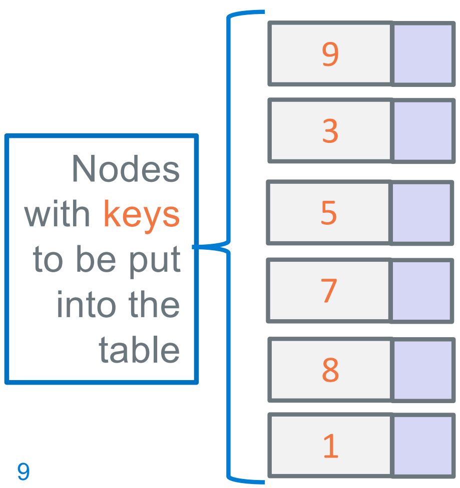
How do you determine on which linked list an entry is stored?



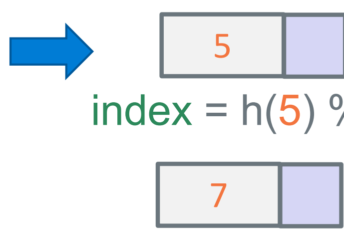
x

Hashing with chaining

- Hash table is an array of pointers to the heads 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***
- Next **use the Hash value** to map to **one entry** in the hash table T[ ] of size TABLESIZE
  - Index = *h*(*k*) % 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



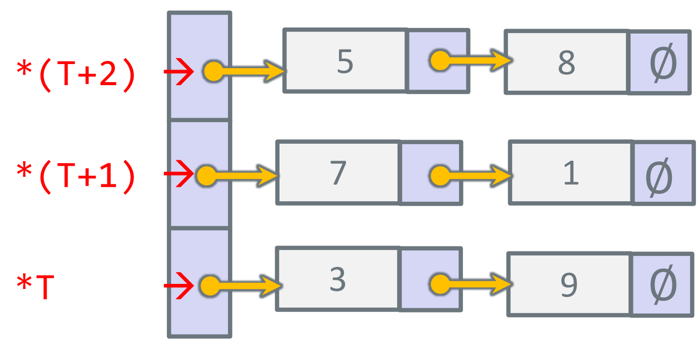
Using a Simple hash
h(k) = k // this is has value
index = k % size // map to a chain



index = h(5) % 3 = 2 -> T+2

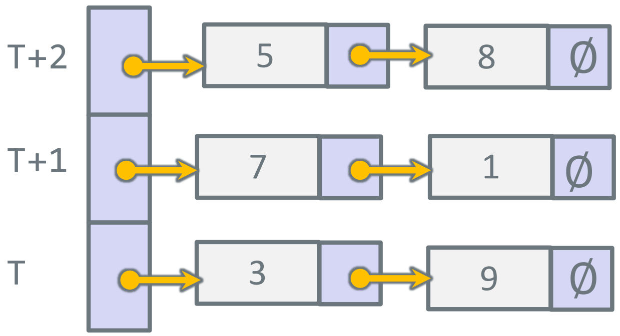
index = h(7) % 3 = 1 -> T+1

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



x

Hash Table With Collision Chaining (multiple linked lists)



Hash Table T[3]
TABLESIZE = 3

- 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 the hash value: hashval = hash(key);
2. Calculate index `i = hashval % TABLESIZE`
3. Go to array element `i`, i.e., `T[i] or *(T+i):` contains the head pointer for collision chain
4. Walk the linked list for element, add element, remove element, etc. from the linked list
5. 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**)*

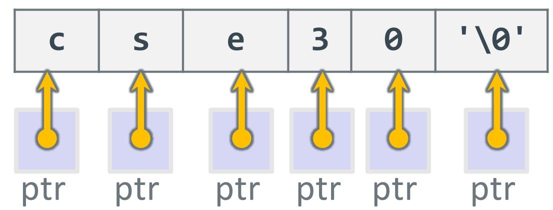
- **Hash arrays** need an **index number to select a chain**, so if we have a string, we must first convert to a hashval number

x

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)

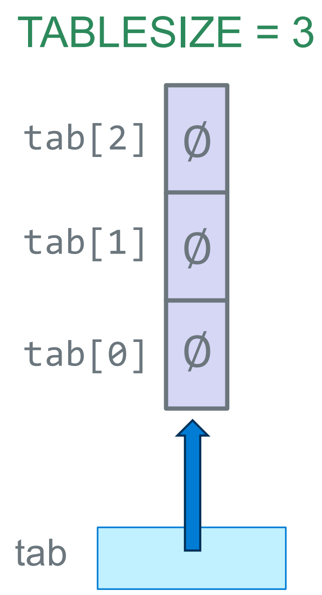


- Many different algorithms for string hash function (Dan Berman's djb2 above)
  - << is the **left** bit shift operator (later in course)
- **Fast to compute**, has a reasonable key distribution for short length ASCII strings into 32-bit unsigned ints

x

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
```

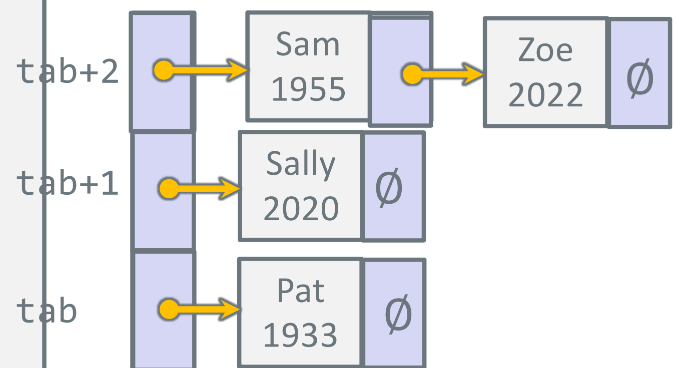


x

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;
```

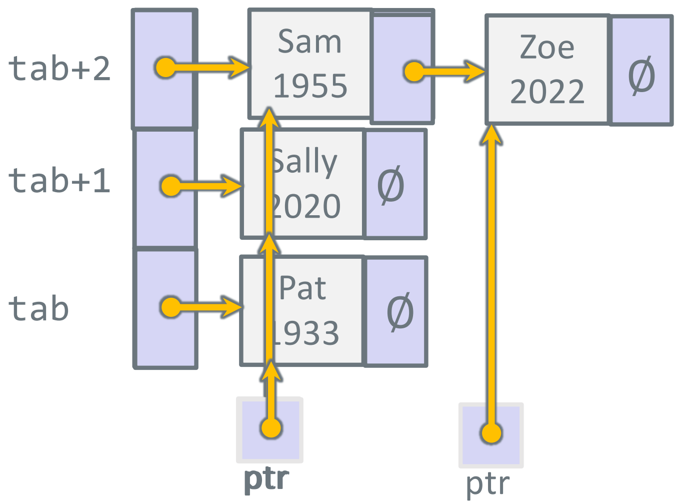
```
struct node {
  int year;
  char *name;
  struct node *next;
};
```



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

x

"Dumping" the Hash Table (traversing all Nodes)

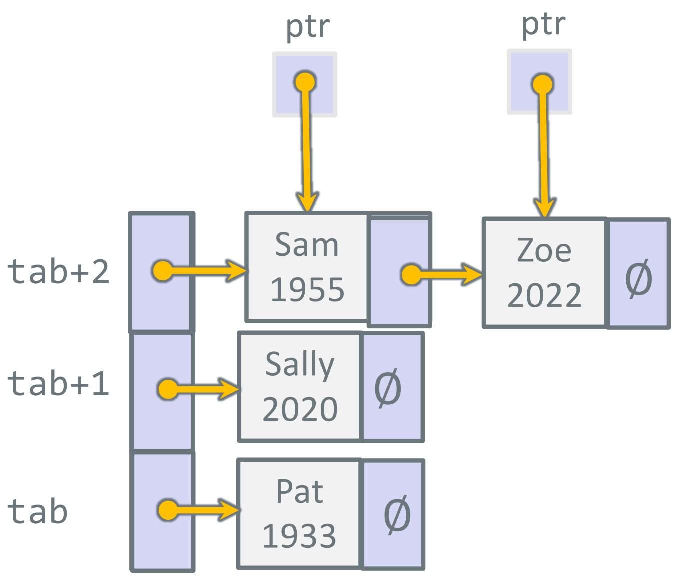



```
Dumping All Data
chain: 0
Year: 1933 Name: Pat
chain: 1
Year: 2020 Name: Sally
chain: 2
Year: 1955 Name: Sam
Year: 2022 Name: Zoe
```

```
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;
    }
}
```


x

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;
}
```

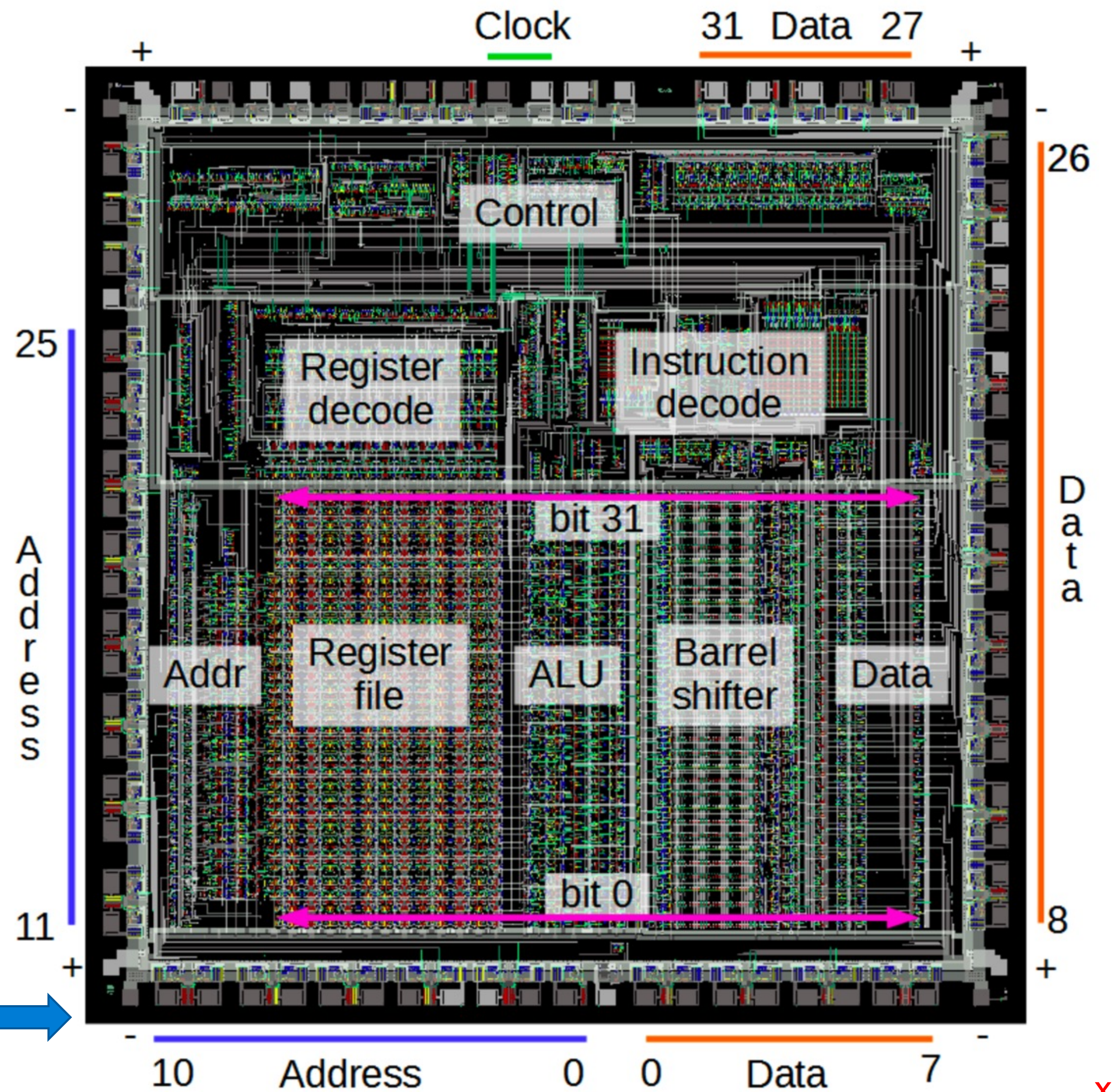
```
index = hash("Zoe") % TBSZ;
if ((ptr = findNode("Zoe", *(tab + index))) != NULL)
    printf("Found Year: %d name: %s\n", ptr->year, ptr->name);
else
    printf("Not Found Zoe\n");
```


X

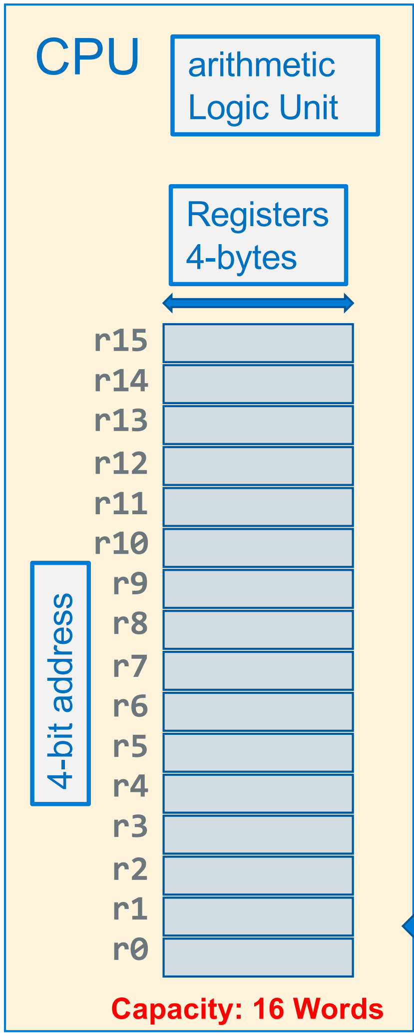
Arm Core Floorplan

- **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***



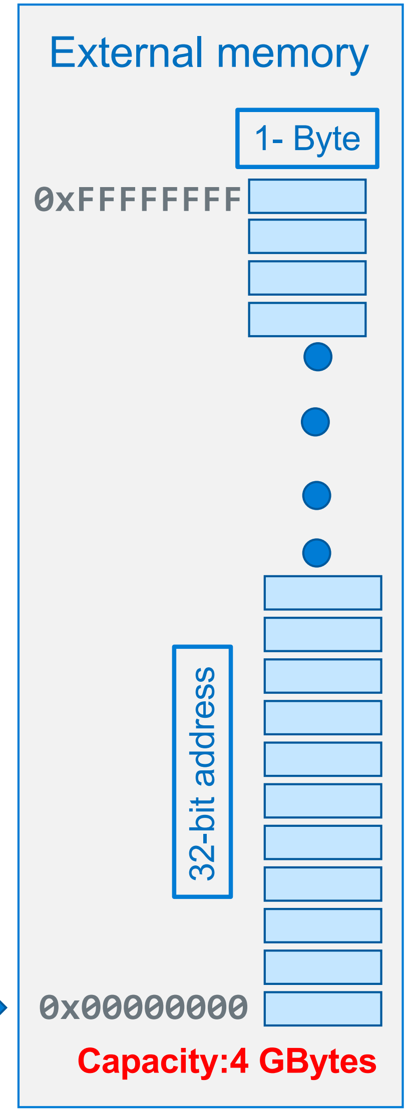
32-Bit Arm - Registers



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

CPU Memory Bus consists of two parts:

Address bus + Data bus



X

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: have a dedicated hardware use (r15 the pc) or special use when used with certain instructions (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

r10
r9
r8
r7
r6
r5
r4

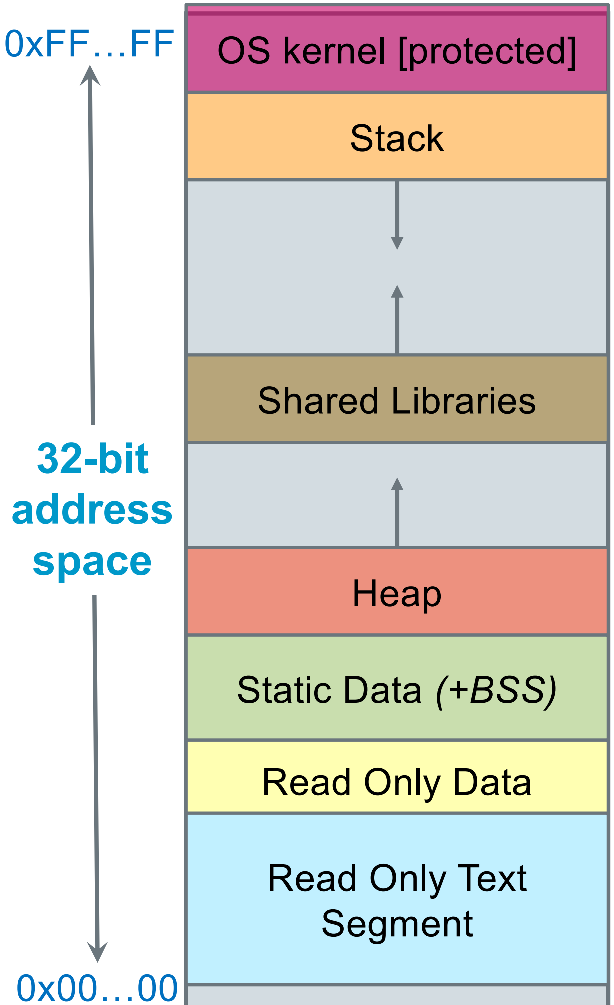
Preserved registers
Called functions can't change

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

r3
r2
r1
r0

x

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:	ebfffb3	bl 102e8 <printf>
10418:	e3a00000	mov r0, 0
1041c:	e24bd004	sub sp, fp, 4

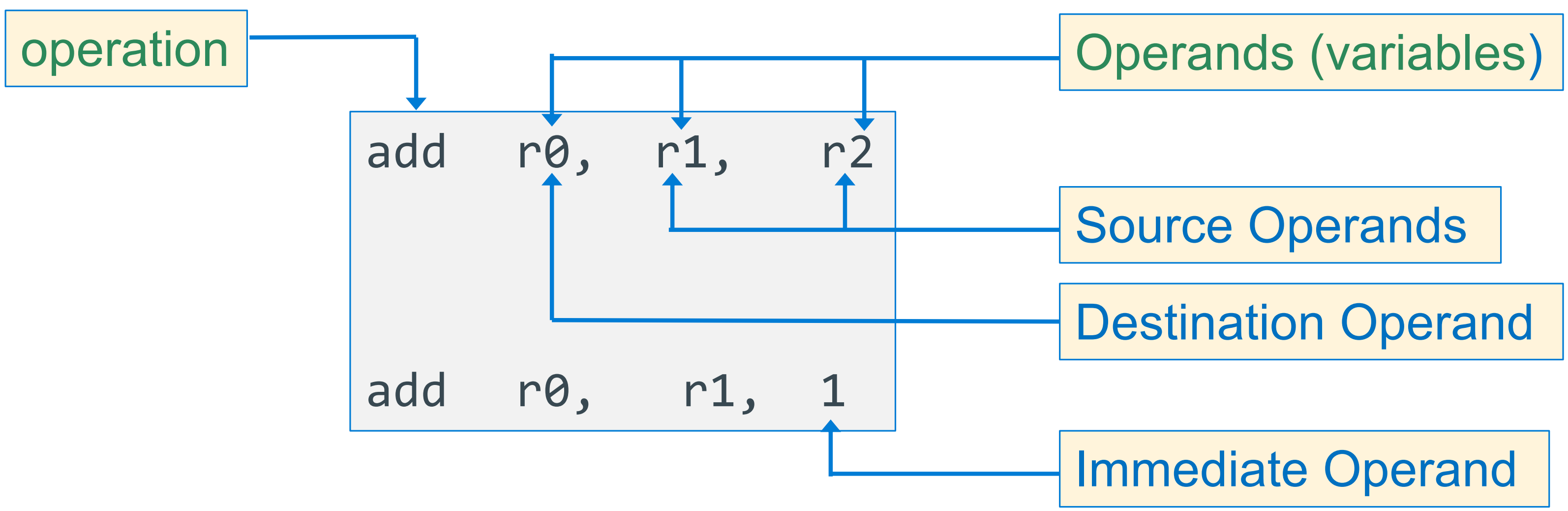
Machine Code

high <- low bytes

x

Anatomy of an Assembly instruction (3 *address* instruction)

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



x

Meaning of an Instruction - ARM

- Operations are abbreviated int opcodes (1– 5 letters)
- Assembly Instructions are specified with a rigid 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
```

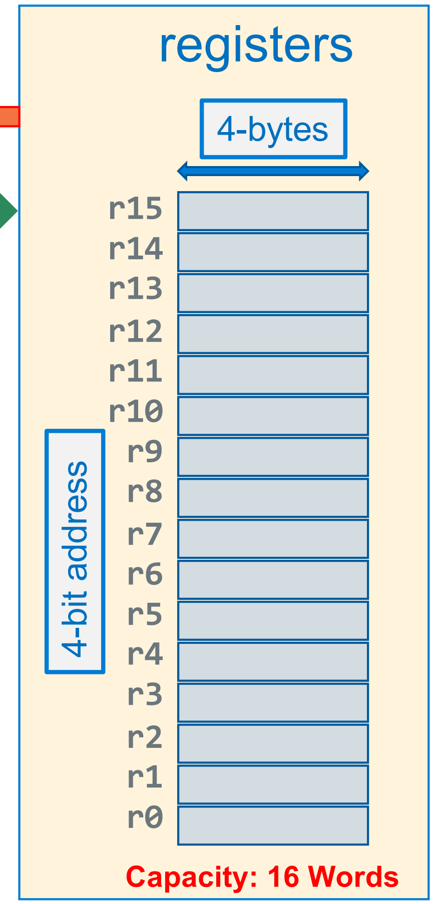
```
              r0 = r1   + r2
    add       r0,  r1,    r2    // assembly
```


x

32-Bit Arm - Registers

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

Arithmetic & Logic Unit (ALU)



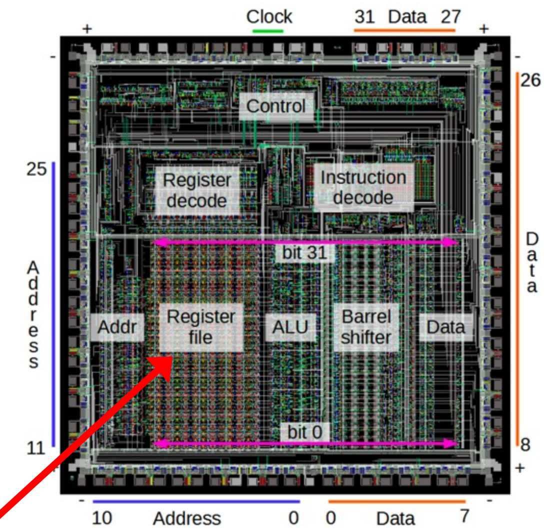
- Almost all arithmetic, logic operations and data movement operations involve at least one register
- As a result, Register addresses are **directly** encoded into 4-bit fields in machine instructions (see below)

31 … 28	27 … 20	19 … 16	15 … 12	11 … 4	3 … 0
Condition	OP code	Rn	Rd	Other info	Rm

x

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

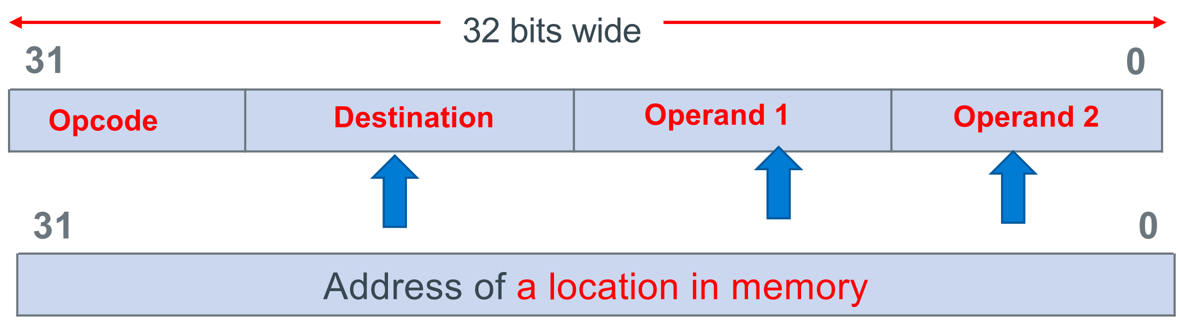
Register contents inside the CPU

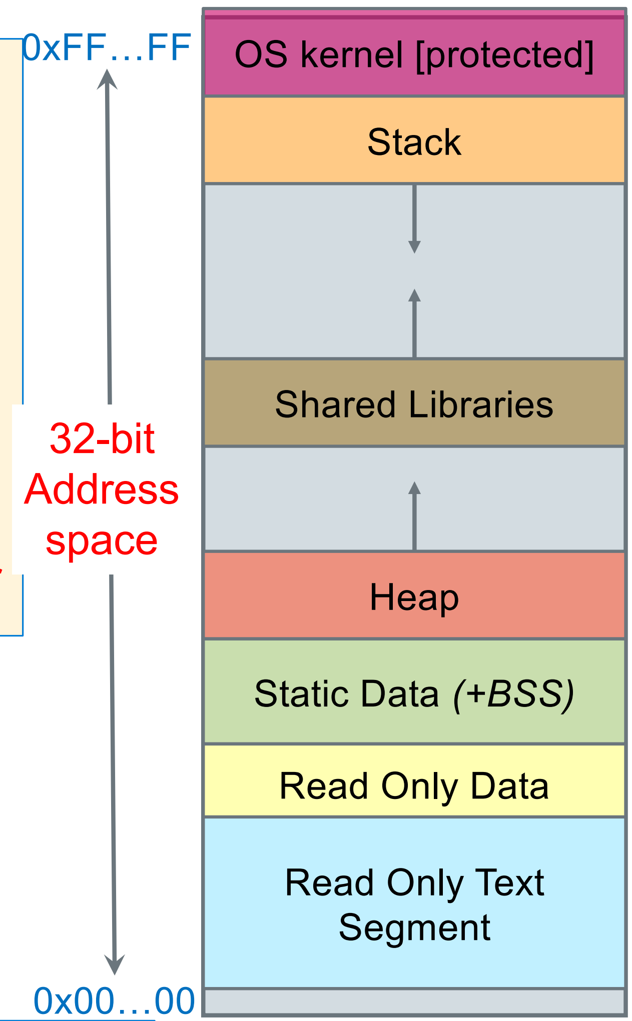
`r0 = 1 r1 = 2`	initial values
`r0 = 4 r1 = 2`	
`r0 = 8 r1 = 2`	
`r0 = 16 r1 = 2`	
`r0 = 16 r1 = 14`	

x

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

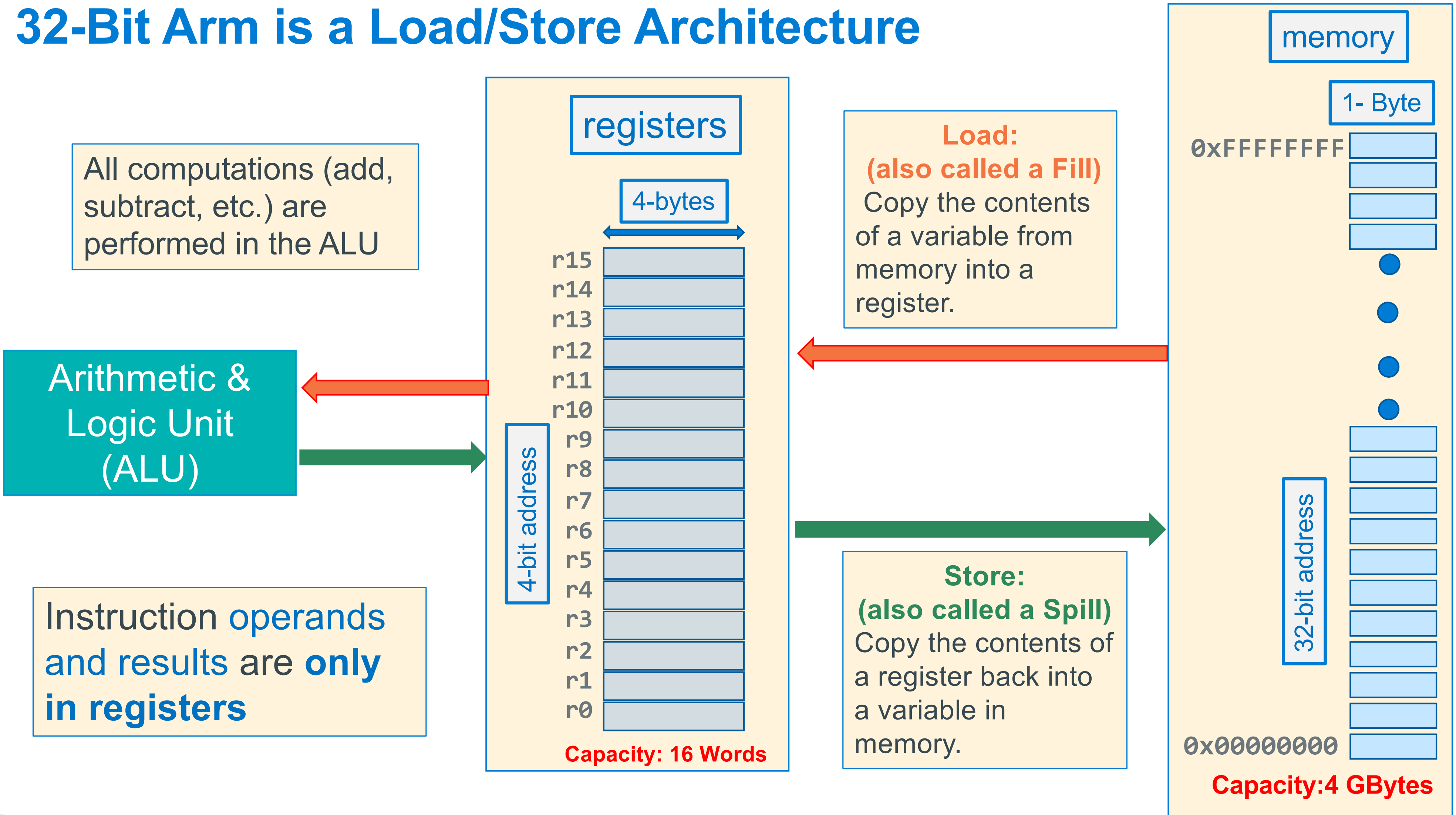




NOT ENOUGH BITS for FULL Addresses to be stored in the instruction

x

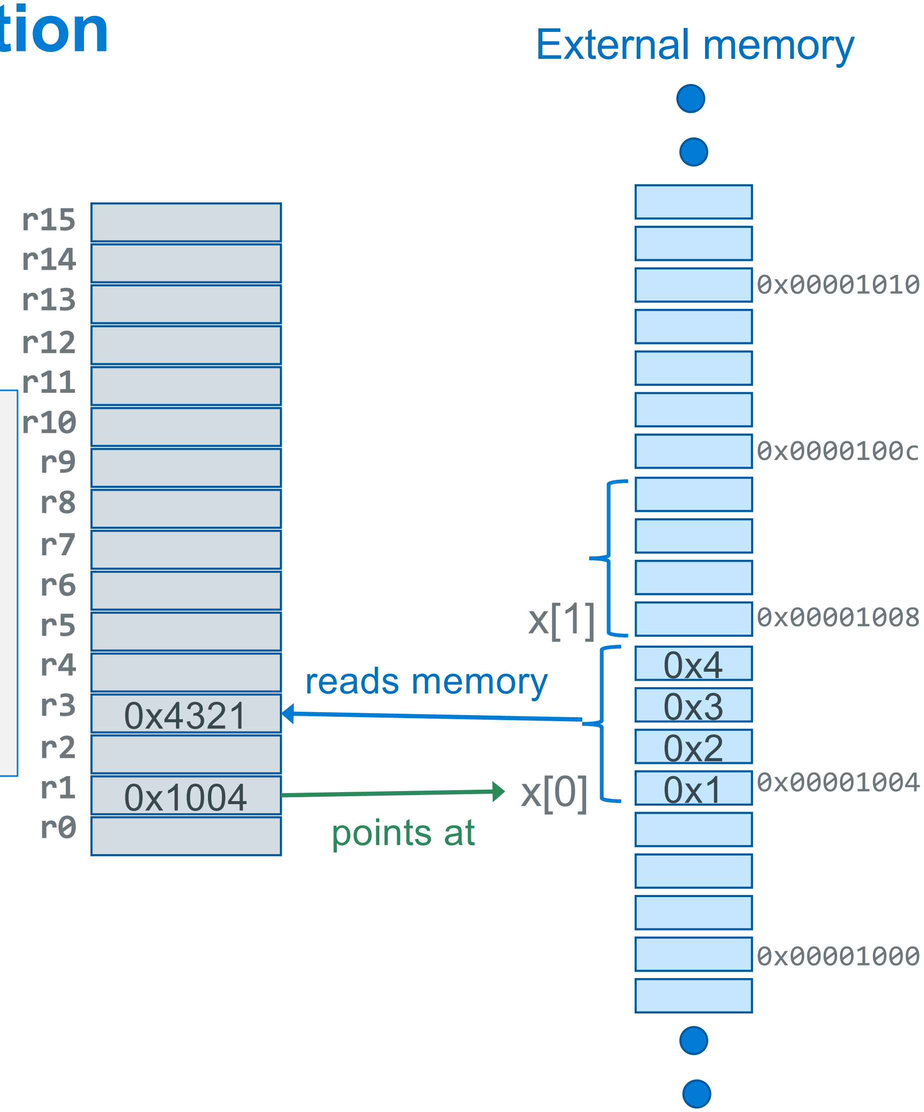
32-Bit Arm is a Load/Store Architecture



Load/Store Concept: Load Operation

```
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
```

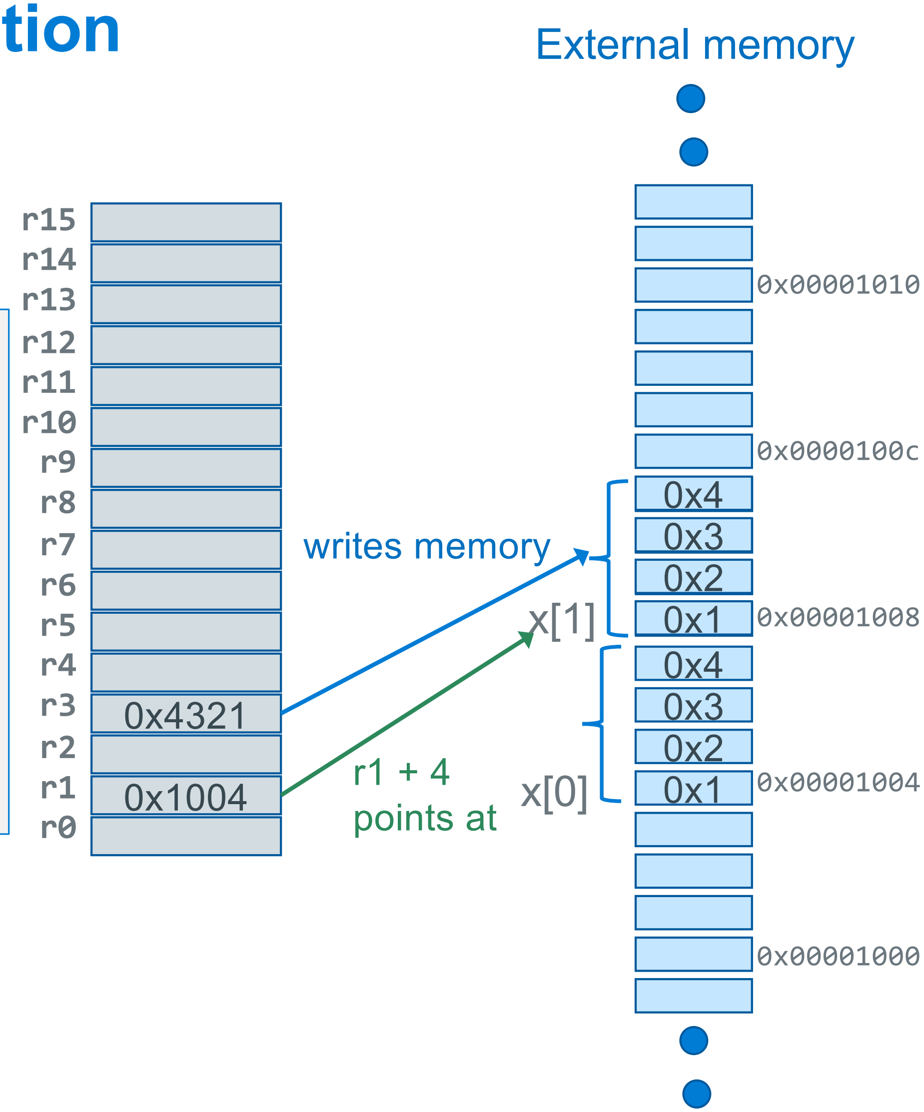


x

Load/Store Concept: Store Operation

```
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
```



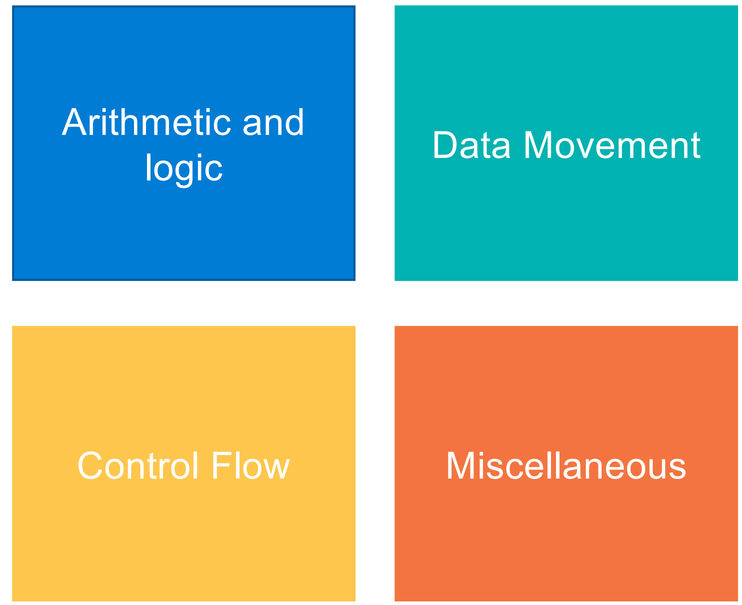
Arm Register Summary

- 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

x

AArch32 Instruction Categories

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

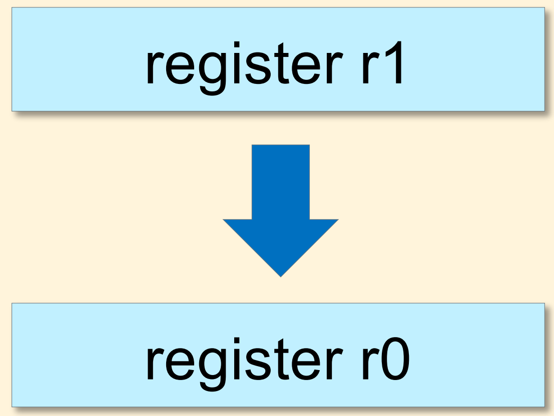


x

First Look: Copying Values Between Registers - MOV

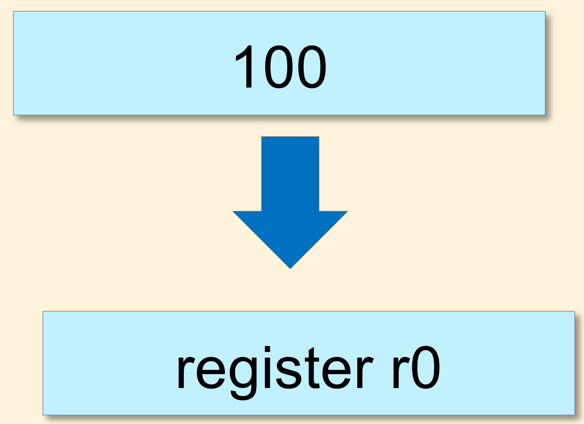
```
mov   r0, r1

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

register direct "addressing"




```
mov   r0, 100

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

Immediate "addressing"



x

mov – Copies Register Content between registers

machine code bit fields

mov	Rd	rot4	imm8

destination register (Rd)
constant (immediate value) (rot4, imm8)

Immediate "addressing"

mov	Rd	Rm

destination register (Rd)
source register (Rm)

register direct "addressing"

	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

Opcode:
1101 - MOV
1111 - MVN

imm8 is 8 bits in size!

IMPACT: limited range of values

(**-256**) <= imm8 <= 255 + values from "rotating" bits - later

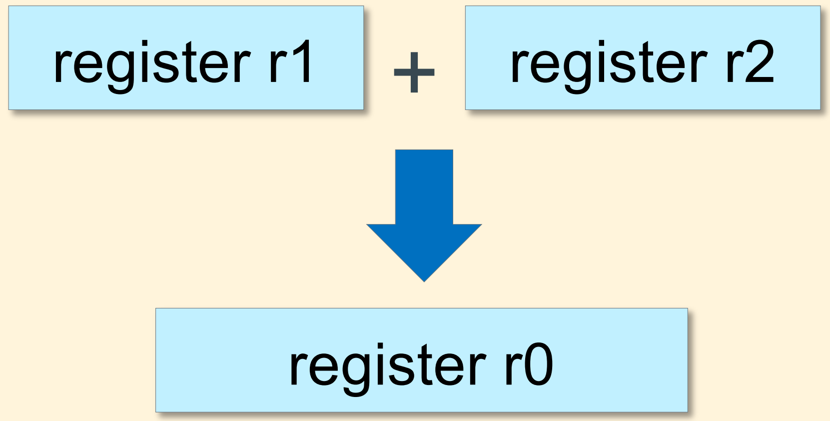
```
mov   Rd, constant    // Rd = constant
mov   Rd,  Rm         // Rd = Rm
```

```
mov  r1, r5              // r1 = r5

mov  r1, 1               // r1 = 1

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

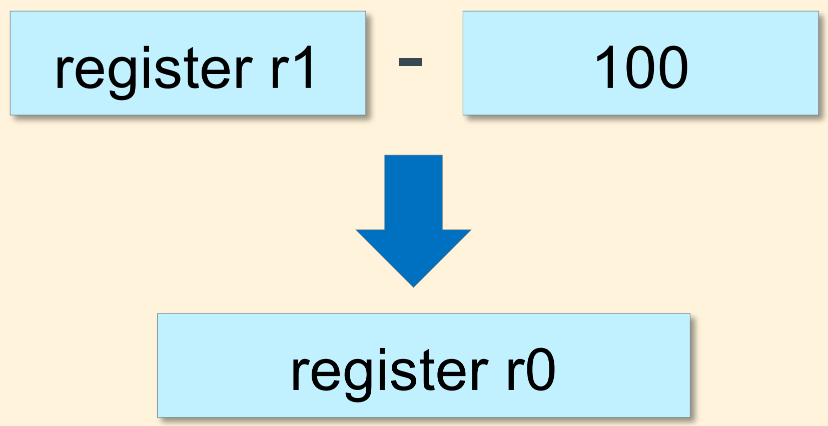

x

First Look: Add/Sub Registers

```
add   r0, r1, r2

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

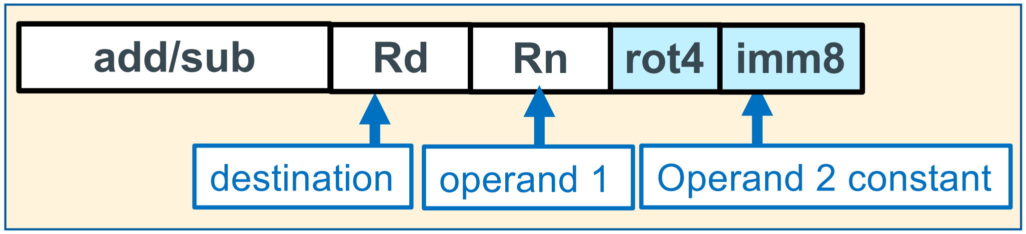


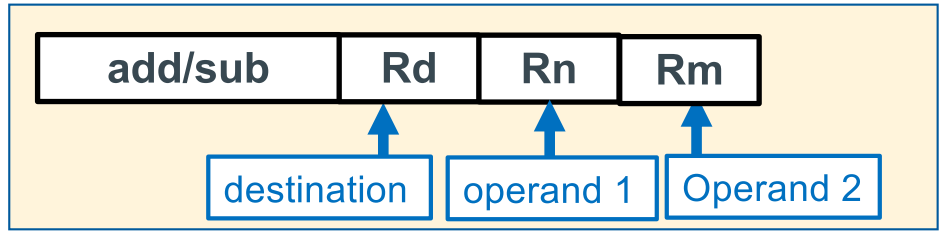

```
sub   r0, r1, 100

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



x

add/sub – 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
```


x

Writing a Sequence of Add & Subtract Instructions

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

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

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

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

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

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

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

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


x

Line Layout in an Arm Assembly Source

column 1 column 2 column 3 column 4

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

tab white space white space

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

x

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 //

x

Labels in Arm Assembly - 2

local label →

Regular label →

local label →

```
.Lmesg: .string  "Hello CSE30! We Are CountinG UpPpER cASe letters!"
main:
        push    {fp, lr}

.Lwhile:
        add     r2, r2, 1       // increment char pointer
```

label .Lmesg is the starting address for the ascii string

label main is the starting address of the push instruction

label .Lwhile is the starting address of the add instruction

- 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";`

x