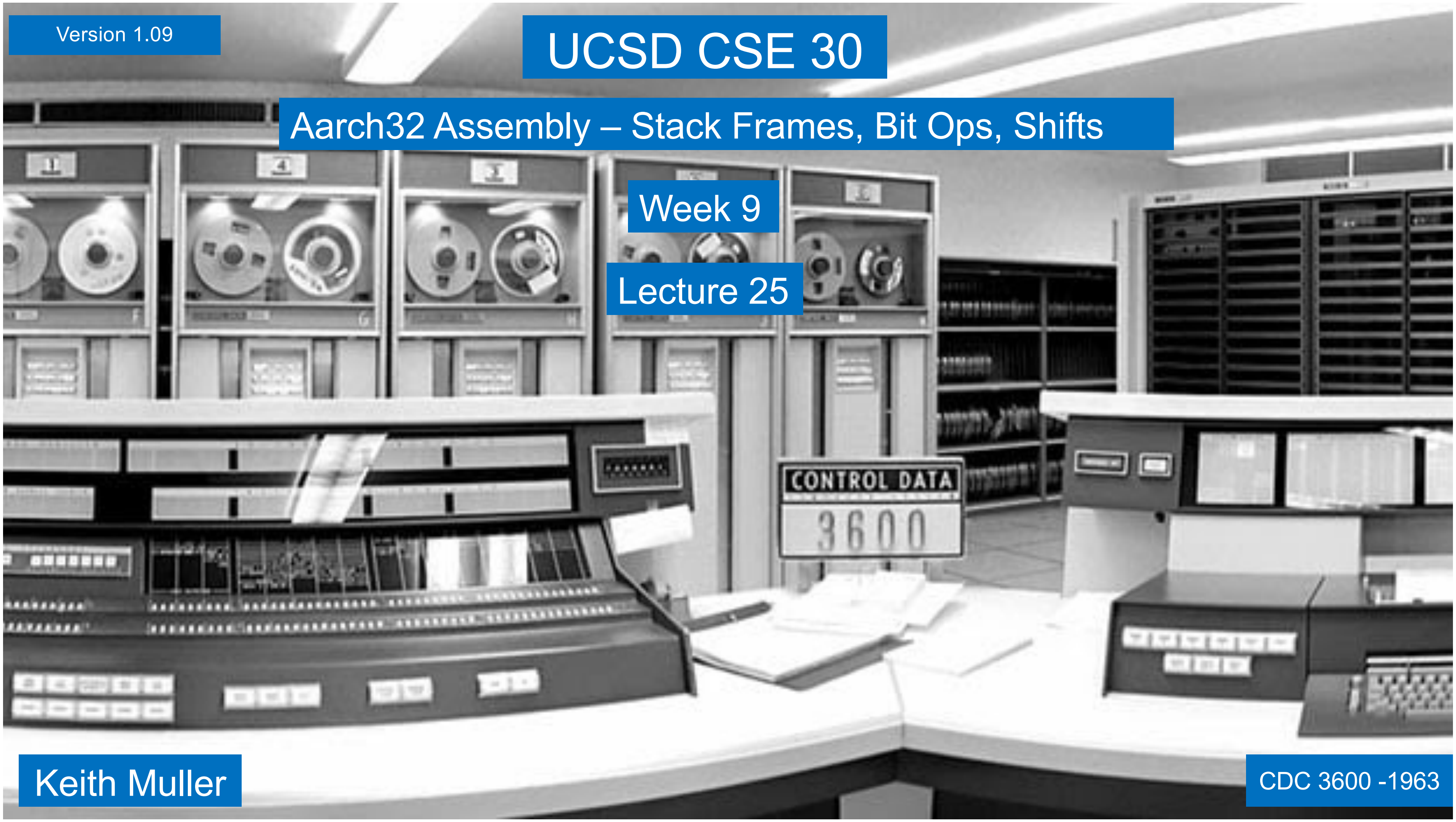
Version 1.09

# UCSD CSE 30

## Aarch32 Assembly – Stack Frames, Bit Ops, Shifts

Week 9

Lecture 25



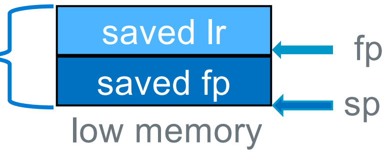
Keith Muller

CDC 3600 -1963

# Function Prologue and Epilogue: Minimum Stack Frame

- **Function prologue** creates stack frame
  1. push/save registers (**`lr`** & **`fp`** minimum) on stack
  2. set **`fp (add fp, …)`** to point at the saved lr as required for use by this function (later)
- **Function epilogue** removes stack frame
  1. set **`sp`** to where it was at the push (we may have **moved sp** to allocate space, later slides)
  2. pop/restore registers (**`lr`** & **`fp`** minimum) from stack
- In this example fp is 4 bytes from sp, (FP_OFF) but this will vary…

smallest frame has:
4 bytes between sp & fp



fp must point at saved lr

```
         .global one
         .type   one, %function
         .equ    FP_OFF,  4

one:
          push    {fp, lr}
          add     fp, sp, FP_OFF

         // your code

          sub     sp, fp, FP_OFF
          pop     {fp, lr}

          bx      lr  // func return

         .size one, (. - one)
```

**Function Header**
Assembly directives

Position the fp

**Function Prologue**
always at top of function

Restore the sp

**Function Epilogue**
always at bottom of function

**Function Footer**
Assembly directive

x

# Stack Creation Overview

1. Calculate how much additional space is needed by local variables
2. **After the push, Subtract from the sp** the required byte count (+ padding - later slides)
3. If the variable has an initial value specified: add code to set the initial value
   a) mov and str are useful for initializing simple variables
   b) loops of mov and str for arrays

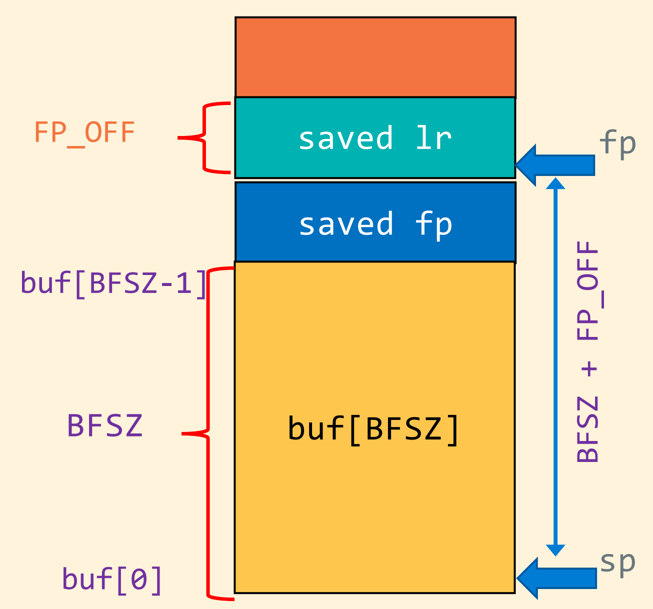
```
#define BFSZ 256
int main(void)
{
  char buf[BFSZ]; // BFSZ bytes
...
```

```
        .equ     FP_OFF, 4
        .equ     BFSZ, 256

main:
        push     {fp, lr}
        add      fp, sp, FP_OFF
        sub      sp, sp, BFSZ
```

**Function Prologue Extended**

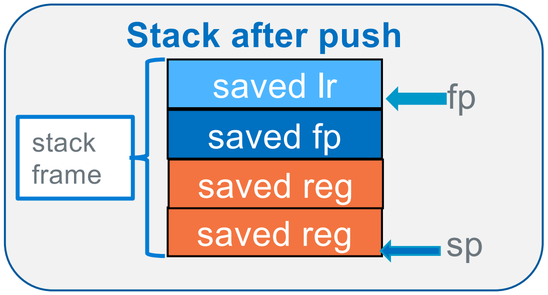
allocate space for buf[256]

stack after allocating local space After sub sp, sp, BFSZ



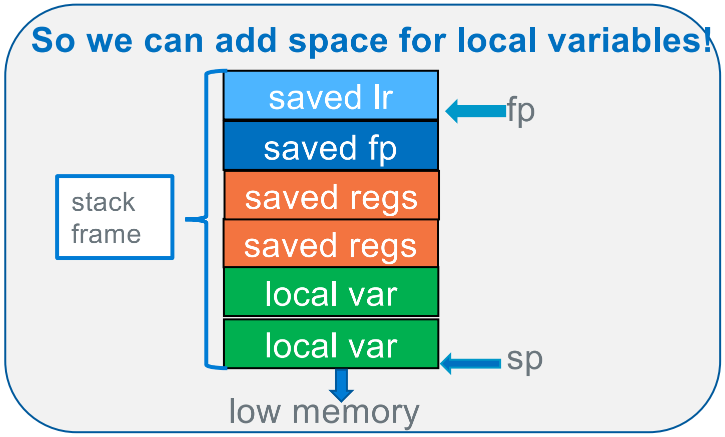
x

# Why is there a `sub, fp, FP_OFF ?`


Stack after push

```
push     {fp, lr}
add      fp, sp, FP_OFF
```

- As you will see, we will move the sp to allocate space on the stack for local variables and parameters, so for the pop to restore the registers correctly:
- sp must point at the last saved preserved register put on the stack bay the save register operation: the push


So we can add space for local variables!

```
   .equ     FRMSZ, 8
   push     {fp, lr}
   add      fp, sp, FP_OFF
   sub      sp, sp, FRMSZ
  // your code

   sub      sp, fp, FP_OFF
   pop      {fp, lr}

   bx       lr  // func return
```

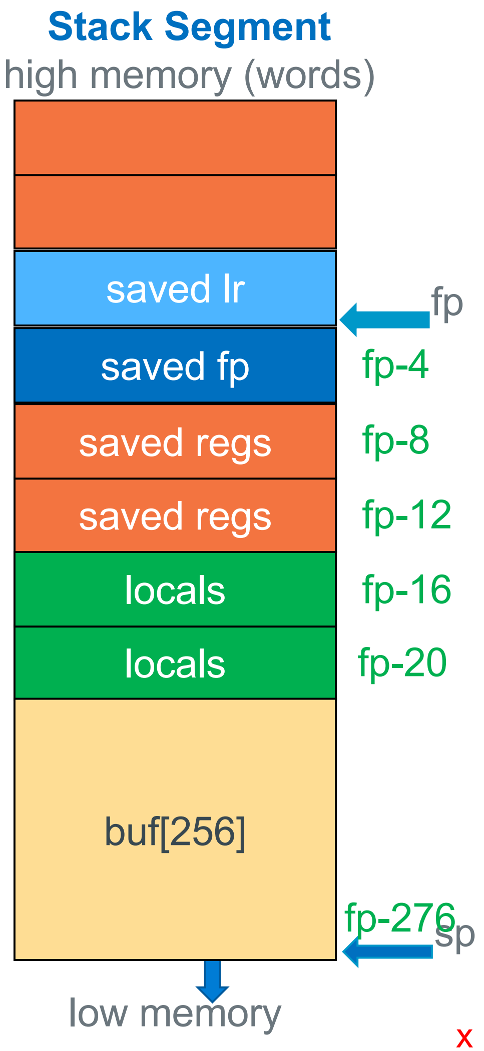
- force the **sp** (using the **fp**) to contain the same address it had after the push operation
  **sub sp, fp, FP_OFF**

x

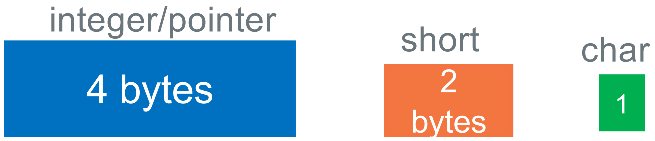
# Accessing the Stack Variables Overview

- Access data stored in the stack use **`ldr/str`** instructions
- **Use base register `fp` with offset addressing** (either register offset or immediate offset)
- No matter where the stack frame starts on the stack, **`fp`** always points at the same place in every stack (points at saved **`lr`**)
- *"hand calculated offset constants and sizes"* like -16 and -20 to access items is easy to get wrong, there is an easier way!

```
.equ  BFSZ, 256          // char array
ldr   r0, [fp, -16]
str   r3, [fp, -20]
sub   r0, fp, 256+20    // r0 = &(buf[0]);
ldr   r1, [r0]          // r1 = buf[0];
str   r3, [r0, 2]       // buf[2] = r3;
```



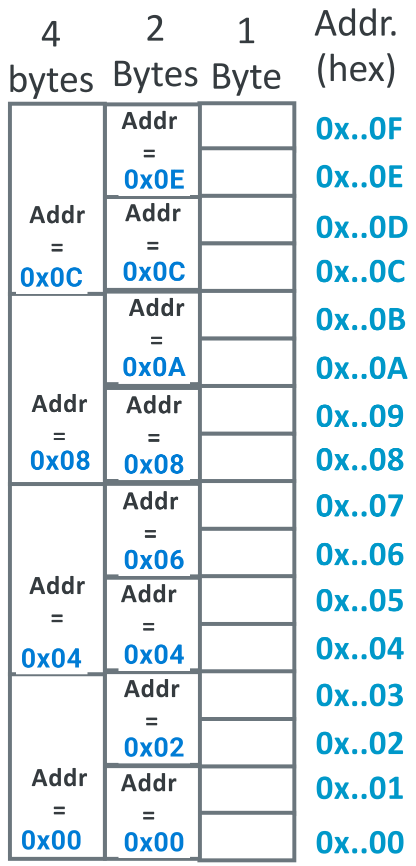
# Variable Alignment on Stack



| Variable Type/Size | Address ends in |
|---|---|
| 8-bit char -1 byte | 0b..0 or 0b..1 |
| 16-bit int -2 bytes | 0b..0 |
| 32-bit int -4 bytes | 0b..00 |
| 32-bit pointer -4 bytes | 0b..00 |

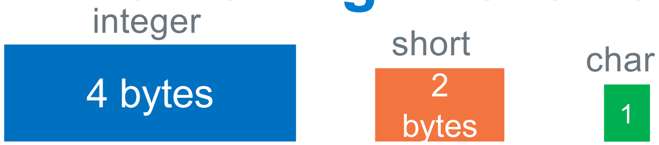
- Starting address alignment requirements for local variables stored on the stack is just like static variables
- sp must be aligned to 8-bytes at function entry & exit
  - contents of sp always ends in `0b..000` at function entry
- Approach we will take (also what compilers often do): allocate all the local variable space as part of the function prologue
  - Aside: You cannot use .align as assembly directives are for fixed address

Starting address by size

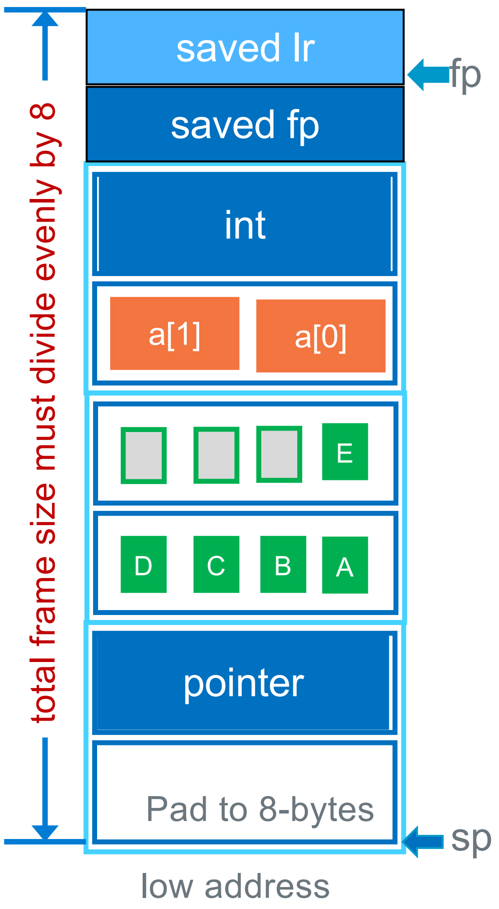


x

# Overview: Stack Frame Alignment Rules



- Goal: minimize stack frame size
- Arrays start at a 4-byte boundary (even arrays with only 1 element)
  - Exception: double arrays [ ] start at an 8-byte boundary
  - struct arrays are aligned to the requirements of largest member
- Space padding when necessary is added at the high address end of a variables allocated space, so the next variable is aligned
- Single chars (and shorts) can be grouped together in same 4-byte word (following the alignment for the short)
- After all the variables have been allocated, add padding at stack frame bottom (low memory) so the total stack frame size (including all saved registers) is a multiple of 8 when the prologue is finished



x

# Stack Frame Design – Step 1 Listing the Local Variables

```
int func(void)
{
    char str[] = "Hi";
    char *ptr = str;
    short buf[3];
    // other code
    int n = 0;
    // other code

    return EXIT_SUCCESS;
}
```

| Variable name | Initial Value | Size bytes | Alignment pad to next | Total Size |
|---|---|---|---|---|
| char str[] | "Hi" | 3 | 1 | 4 |
| char *ptr | str | 4 | 0 | 4 |
| short buf[3] | | 3 * 2 | 2 | 8 |
| int n | 0 | 4 | 0 | 4 |
| <sub total> | | | | 20 |

- Create a table of all the variables defined **throughout the entire function** starting from function start to the end of the function
- For each variable: list its size, initial value if any, alignment padding and sum to total size
  - When needed: padding after the variable (the high address side) to fill out the allocation

# Stack Frame Design – Step 2 Layout the Frame & Size It

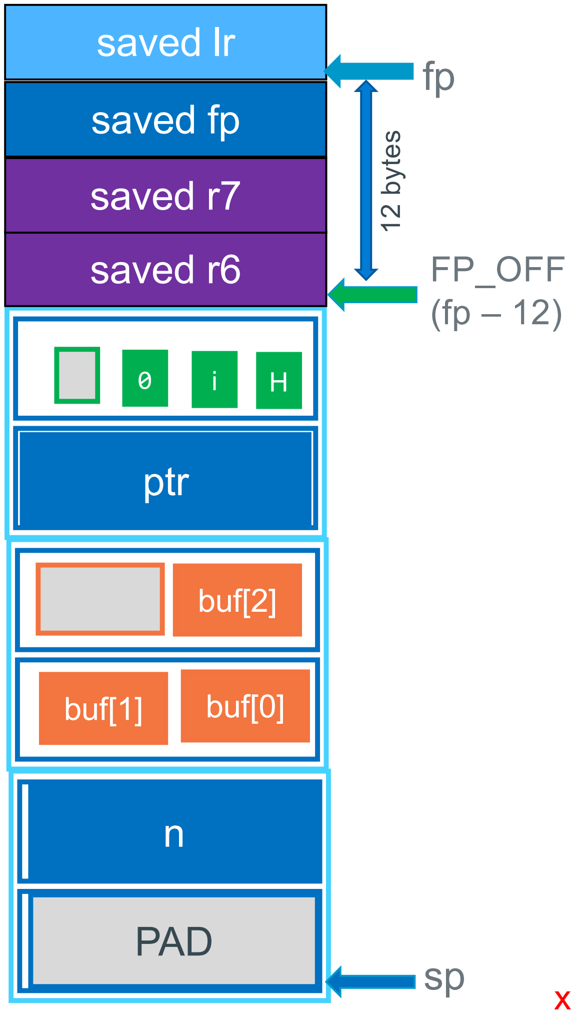
4 bytes

2 bytes

1

| Variable name | Initial Value | Size bytes | Alignment pad to next | Total Size |
|---|---|---|---|---|
| `char str[]` | "Hi" | 3 | 1 | 4 |
| `char *ptr` | str | 4 | 0 | 4 |
| `short buf[3]` | | 3 * 2 | 2 | 8 |
| `int n` | 0 | 4 | 0 | 4 |
| `<sub total>` | | | | 20 |

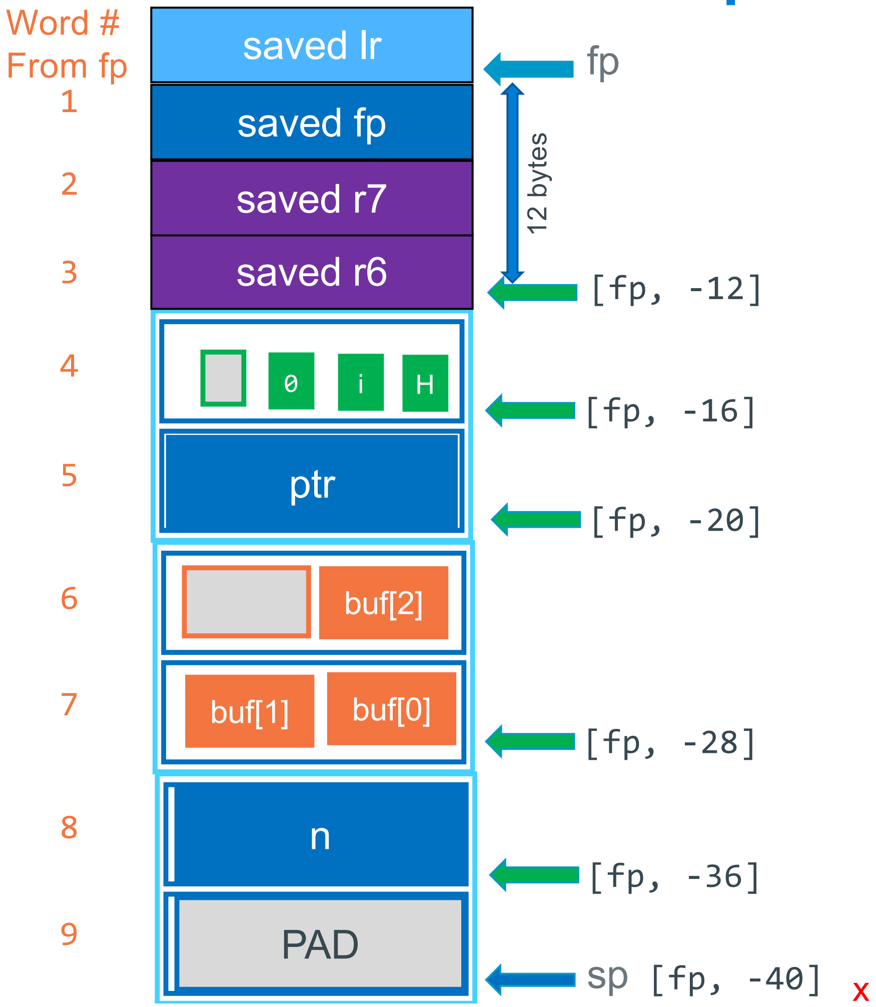
| Allocation Type | Total |
|---|---|
| FP_OFF + 4 = 12 + 4 = 16 | 16 |
| Local Variables Sub total | 20 |
| Space for parameters on stack (later) | 0 |
| total before pad | 36 |
| Pad as needed to align 8-byte boundary | 4 |
| TOTAL Size for entire frame | 40 |



x

# Stack Frame Design – Step 3 Generate the Offsets from fp

- Word offset is a way to visualize the distance from fp for calculating offset values
- Better to have the assembler to generate readable offsets for use with `str` and `ldr`
  1. Easy to add and remove variable allocations from the design
  2. Creates well documented names for each variable: `ldr r0, [fp -20]` is hard to read
  3. Automatically calculates the total size of the stack frame used by local variables



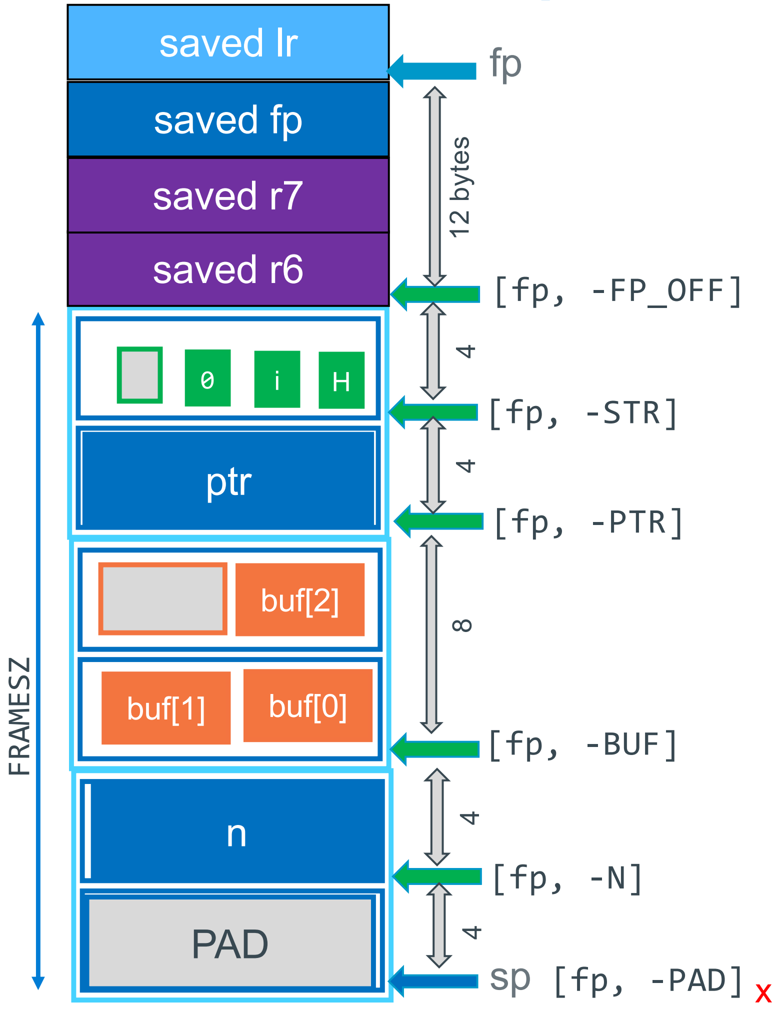
# Stack Frame Design – Step 3 Generate the offsets from fp

| Variable name | Initial Value | Size bytes | Alignment pad to next | Total Size |
|---|---|---|---|---|
| `char str[]` | "Hi" | 3 | 1 | 4 |
| `char *ptr` | str | 4 | 0 | 4 |
| `short buf[3]` | | 3 * 2 | 2 | 8 |
| `int n` | 0 | 4 | 0 | 4 |

| Allocation Type | Total |
|---|---|
| FP_OFF + 4 = 12 + 4 = 16 | 16 |
| Pad to get to 8-byte boundary | 4 |
| FRAMESZ **space for Locals** (PAD – FP_OFF) | 24 |

```
        .equ   FP_OFF,       12  // local base
           // NAME,          SIZE + prev_name
        .equ   STR,          4 + FP_OFF
        .equ   PTR,          4 + STR
        .equ   BUF,          8 + PTR
        .equ   N,            4 + BUF
        .equ   PAD,          4 + N
        .equ   FRAMESZ       PAD - FP_OFF
```

Distance Offsets from fp



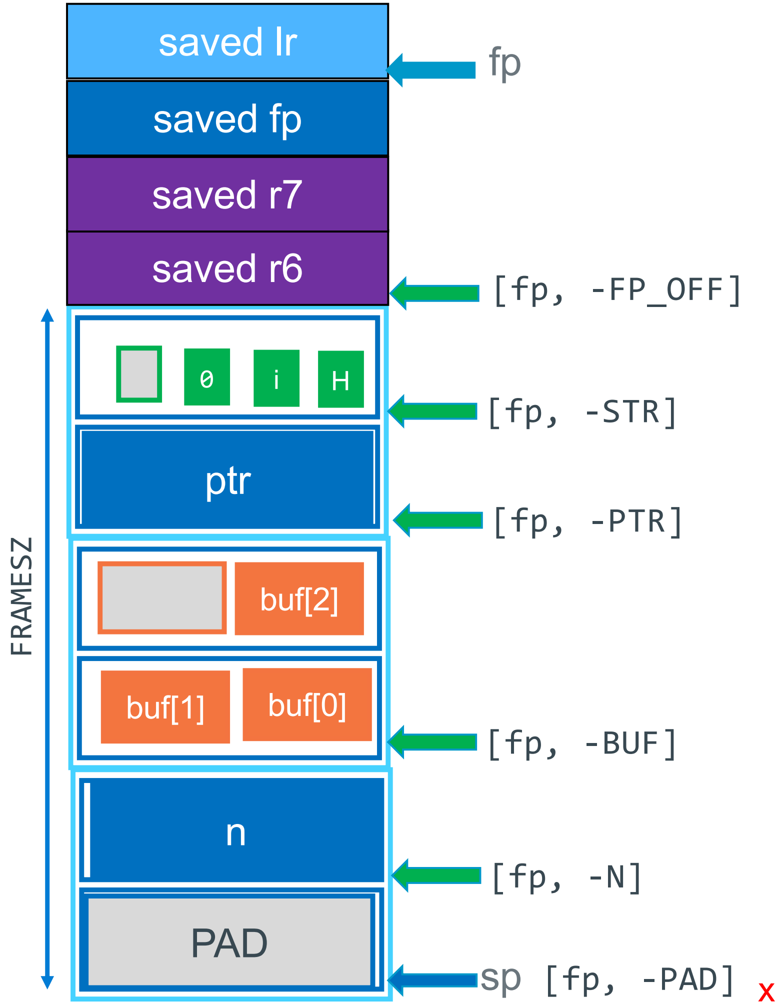
# Stack Frame Design – Step 4 Modifying the prologue

Distance Offsets from fp

```
        .equ   FP_OFF,        12  // local base
          // NAME,            SIZE + prev_name
        .equ   STR,           4 + FP_OFF
        .equ   PTR,           4 + STR
        .equ   BUF,           8 + PTR
        .equ   N,             4 + BUF
        .equ   PAD,           4 + N
        .equ   FRAMESZ        PAD - FP_OFF
```

```
main:
   push     {r6, r7, fp, lr}
   add      fp, sp, FP_OFF
   sub      sp, sp, FRAMESZ // add for locals
 ➡  // no change to epilogue ⬅
```

| *variable* | *arm ldr/str statement examples* |
|---|---|
| n | `ldr/str     r0, [fp, -N]` |
| buf[1] | `ldrh/strh   r0, [fp, -BUF + 2]` |
| &(str[0]) | `sub     r0, fp, STR` |

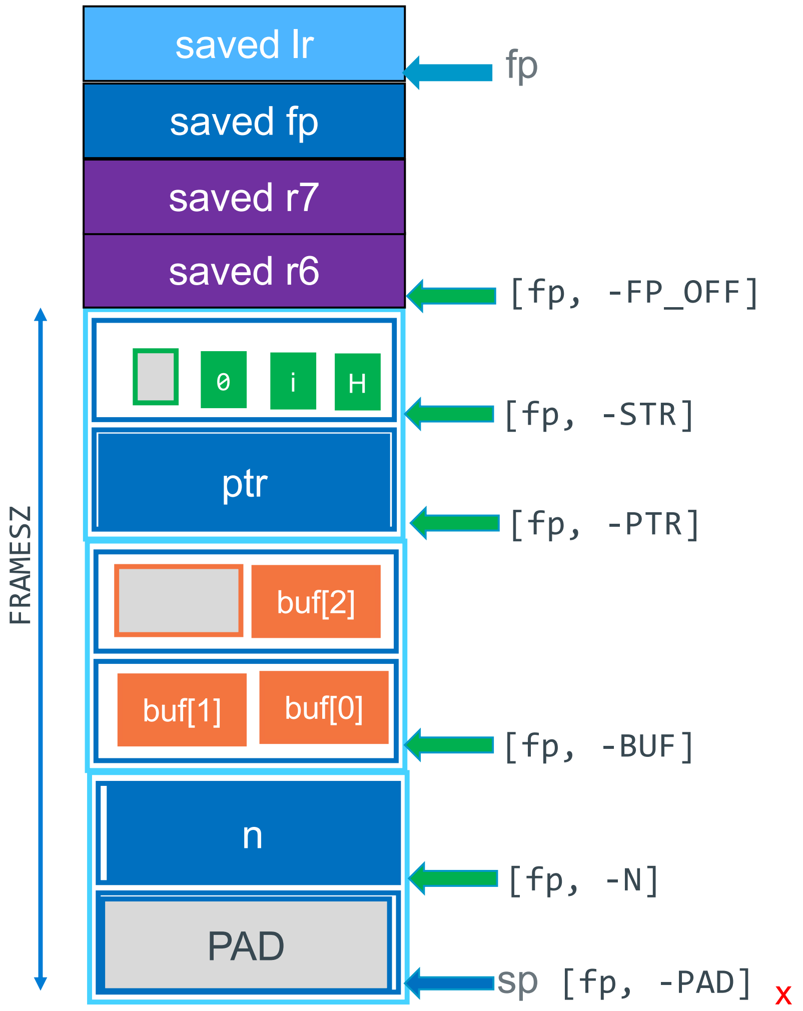


# Stack Frame Design – Step 5 Initialize the variables

```
char str[] = "Hi";
char *ptr = str;
short buf[3];
// other code
int n = 0;
// other code
```

Used in PA5

```
main:
       push    {r6, r7, fp, lr}
       add     fp, sp, FP_OFF
       sub     sp, sp, FRAMESZ
       sub     r6, fp, STR      // &(str[0])
       str     r6, [fp, -PTR]
       mov     r6, 'H'
       strb    r6, [fp, -STR]
       mov     r6, 'i'
       strb    r6, [fp, -STR+1]
       mov     r6, 0
       strb    r6, [fp, -STR+2]
       // other code
       mov     r6, 0
       str     r6, [fp, -N]
```



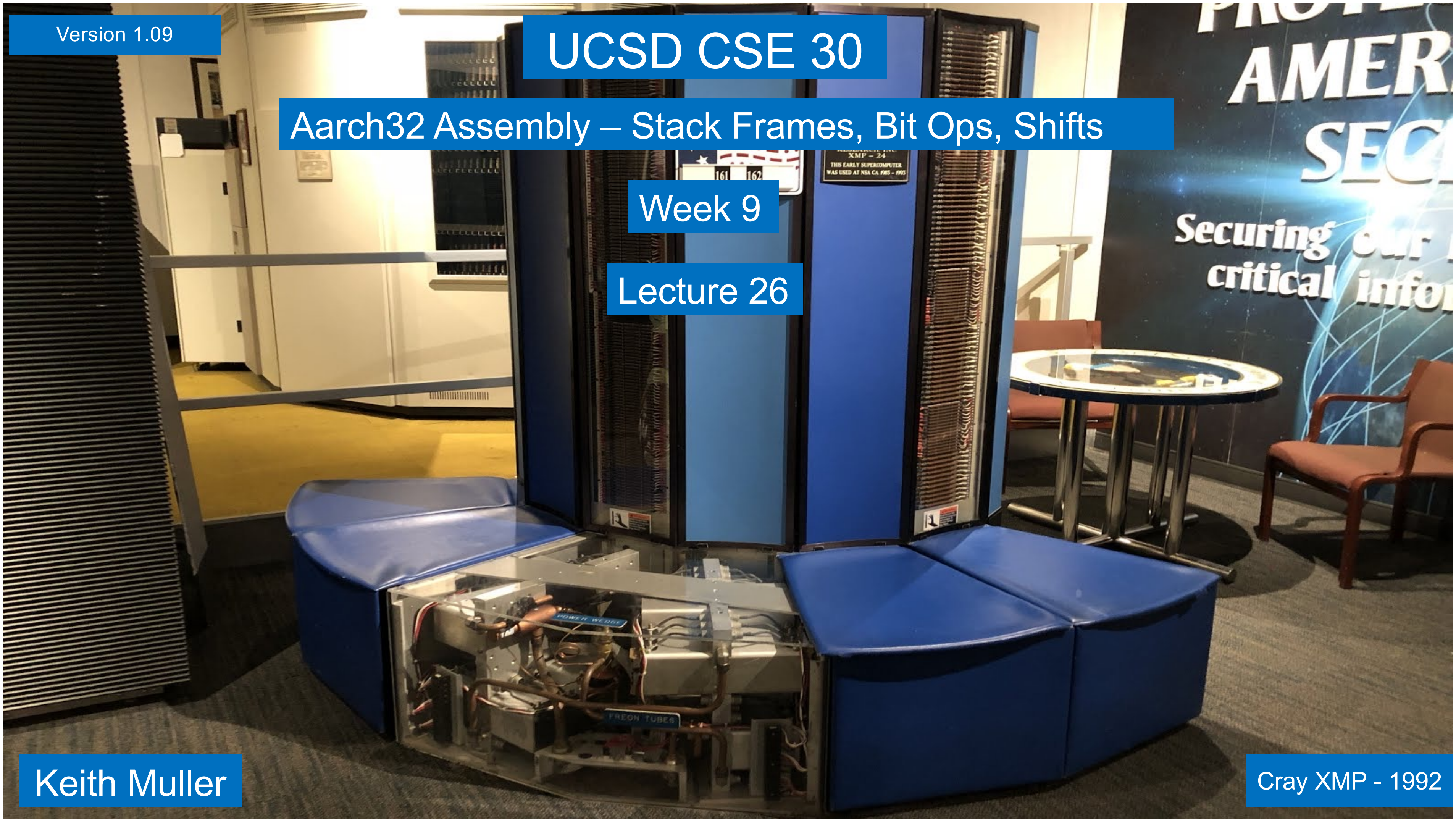
Version 1.09

# UCSD CSE 30

## Aarch32 Assembly – Stack Frames, Bit Ops, Shifts

Week 9

Lecture 26



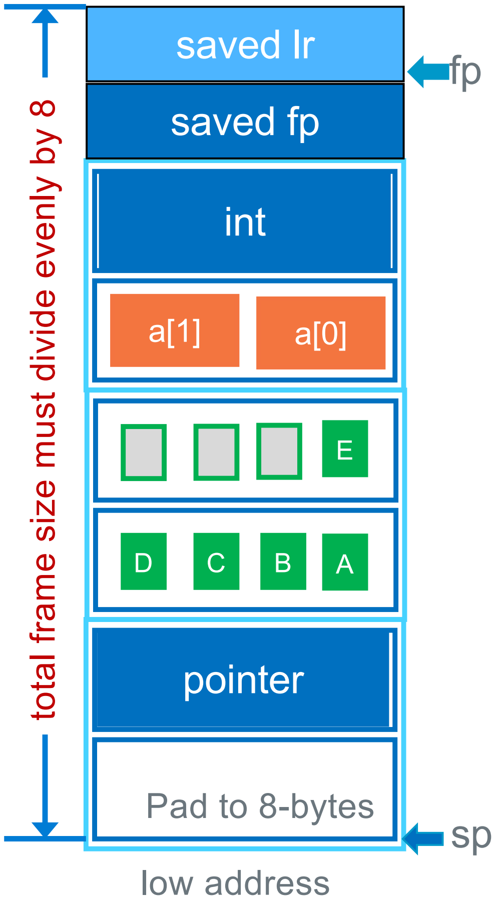
Keith Muller

Cray XMP - 1992

# Overview: Stack Frame Alignment Rules

integer
4 bytes
short
2 bytes
char
1

- Goal: minimize stack frame size
- Arrays start at a 4-byte boundary (even arrays with only 1 element)
  - Exception: double arrays [ ] start at an 8-byte boundary
  - struct arrays are aligned to the requirements of largest member
- Space padding when necessary is added at the high address end of a variables allocated space, so the next variable is aligned
- Single chars (and shorts) can be grouped together in same 4-byte word (following the alignment for the short)
- After all the variables have been allocated, add padding at stack frame bottom (low memory) so the total stack frame size (including all saved registers) is a multiple of 8 when the prologue is finished



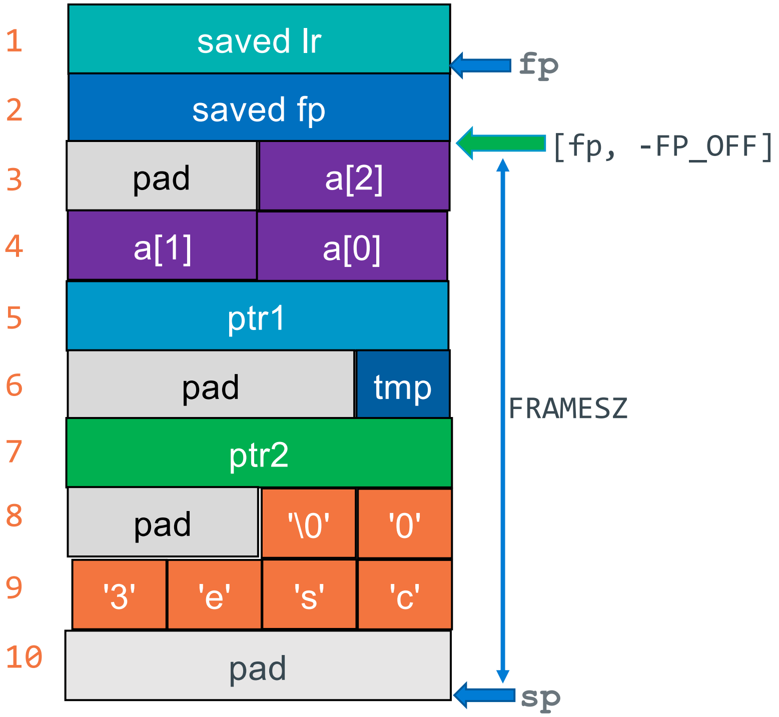
x

# Local Variables: Stack Frame Design Practice

Example shows allocation **without reordering** variables to optimize space

```
    short a[3];
    short *ptr1;
    char tmp;
    char *ptr2;
    char nm[] = "cse30";
```

```
.equ   FP_OFF,       4  // Local base
// NAME,          SIZE + prev_name
.equ   A,            8 + FP_OFF
.equ   PTR1,         4 + A
.equ   TMP,          4 + PTR1
.equ   PTR2,         4 + TMP
.equ   NM,           8 + PTR2
.equ   PAD,          4 + NM
.equ   FRAMESZ       PAD – FP_OFF // for locals
```



**When writing real code, you do not have to put all locals on the stack**

- Place locals in registers if they fit, are accessed often, **and**
- You do not need their address (they are not an output variable in a function call)

x

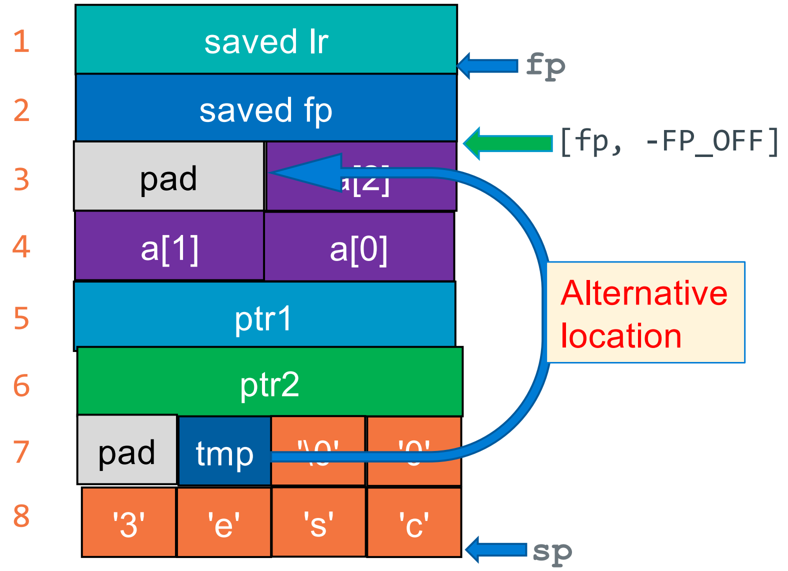
# Local Variables: Stack Frame Design Reordering

Example shows allocation **with reordering** variables to optimize space

```
    short a[3];
    short *ptr1;
    char *ptr2;
    char tmp;
    char nm[] = "cse30";
```

```
.equ   FP_OFF,       4  // Local base
// NAME,            SIZE + prev_name
.equ   A,            8 + FP_OFF
.equ   PTR1,         4 + A
.equ   PTR2,         4 + PTR1
.equ   TMP,          2 + PTR2
.equ   NM,           6 + TMP
.equ   PAD,          0 + NM // not needed
.equ   FRAMESZ       PAD - FP_OFF
```

size change

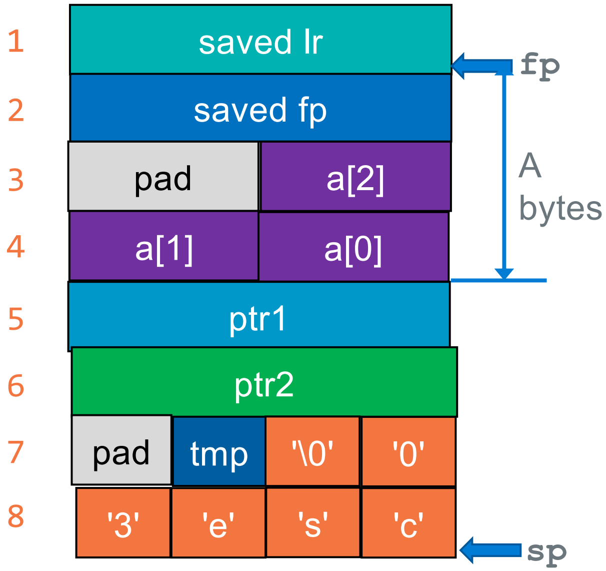


**When writing real code, you do not have to put all locals on the stack**

- Place locals in registers if they fit, are accessed often, **and**
- You do not need their address (they are not an output variable in a function call)

x

# Entire source file



| | *Evaluated into r0* |
|---|---|
| `&(a[1])` | `sub   r0, fp, A - 2` |
| `&(a[1])` | `add   r0, fp, -A + 2` |
| `&(nm[1])` | `add   r0, fp, -NM + 1` |
| `ptr2` | `add   r0, fp, -PTR2` |

```
        .arch armv6
        .arm
        .fpu vfp
        .syntax unified
         // globals etc here
        .text
        .type       doit, %function
        .global     doit
        .equ        EXIT_SUCCESS, 0
        .equ        FP_OFF,    4  // Local base
        .equ        A,         8 + FP_OFF
        .equ        PTR1,      4 + A
        .equ        PTR2,      4 + PTR1
        .equ        TMP,       2 + PTR2
        .equ        NM,        6 + TMP
        .equ        PAD,       0 + NM
        .equ        FRAMESZ    PAD - FP_OFF
doit:
        push    {fp, lr}
        add     fp, sp, FP_OFF
        sub     sp, sp, FRAMESZ
        // doit() code goes here
        mov     r0, EXIT_SUCCESS

        sub     sp, fp, FP_OFF
        pop     {fp, lr}
        bx      lr
        .size doit, (. - doit)
        .section .note.GNU-stack,"",%progbits
.end
```

With large frames you may need to use ldr if the immediate value FRAMESZ does not fit in imm8 (r3 is not a parameter in this example)

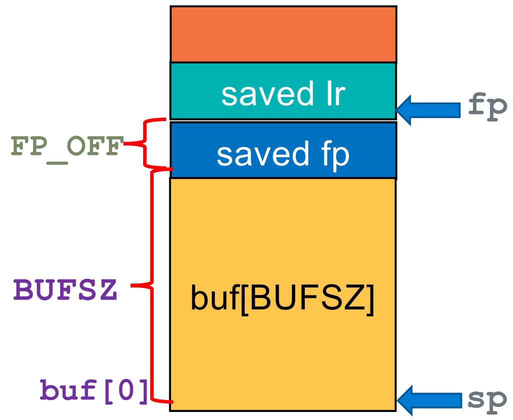
```
ldr     r3, =FRAMESZ
sub     sp, sp, r3
```

x

# Passing an Output Parameter

```
#define BUFSZ 256
int main(void)
{
  char buf[BUFSZ];
  if (fgets(buf, BUFSZ, stdin) != NULL)
    printf("%s", buf);
  return EXIT_SUCCESS;
}
```

```
char *fgets(char *s, int size, FILE *stream);
 returns *s or NULL    r0,       r1,          r2
```



if the immediate value of BUF does not fit in imm8

```
ldr     r0, =BUF
sub     r0, fp, r0
```

if the immediate value of BUFSZ does not fit in imm8

```
ldr     r1, =BUFSZ
```

```
        .extern printf
        .extern fgets
        .extern stdin
        .section .rodata
.Lpfstr .string  "%s"
        .text
         // function header stuff not shown
        .equ    BUFSZ,      256
        .equ    FP_OFF,     4
        .equ    BUF,        BUFSZ + FP_OFF
        .equ    FRAMESZ,    BUF - FP_OFF
main:
        push    {fp, lr}
        add     fp, sp, FP_OFF
        sub     sp, sp, FRAMESZ
        sub     r0, fp, BUF
        mov     r1, BUFSZ
        ldr     r2, =stdin
        ldr     r2, [r2]
        bl      fgets
        cmp     r0, NULL
        beq     .Ldone
        mov     r1, r0
        ldr     r0, =.Lpfstr
        bl      printf
.Ldone: // rest of file not shown
```

stdin is a global variable pointer *FILE

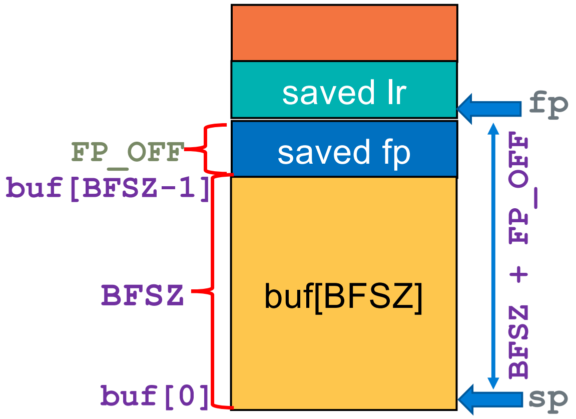
```
r0 = &(buf[0]);
r1 = BUFSZ;
r2 = stdin
```

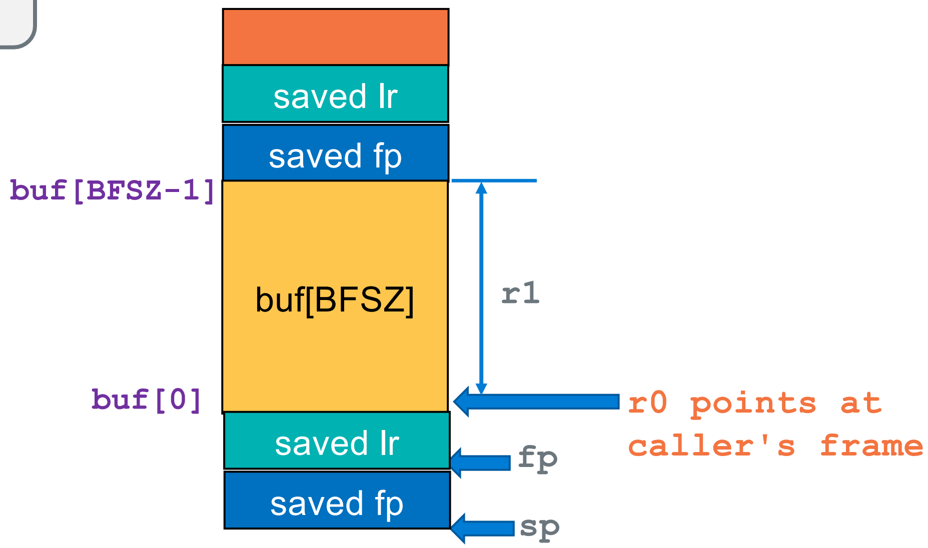
X

# Writing Functions: Receiving an Output Parameter - 1

```
#define BFSZ 256
void fillbuf(char *s, int len, char fill);
int main(void)   r0,        r1,        r2
{
  char buf[BFSZ];
  fillbuf(buf, BFSZ, 'A');
  return EXIT_SUCCESS;
}
```

```
void fillbuf(char *s, int len, char fill)
{    r0,         r1,         r2
    char enptr = s + len;
    while (*s < enptr)
        *(s++) = fill;
}
```



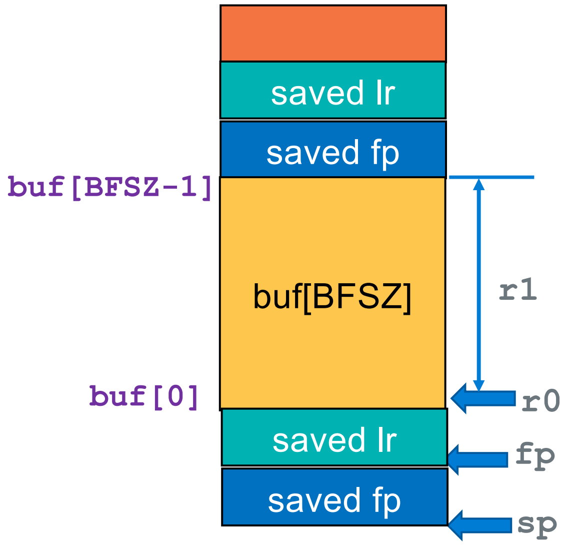


x

# Writing Function: Receiving an Output Parameter - 2

```
void          r0,         r1,        r2
fillbuf(char *s, int len, char fill)
{
    char enptr = s + len;
    while (s < enptr)
        *(s++) = fill;
}
```

Using r1 for endptr



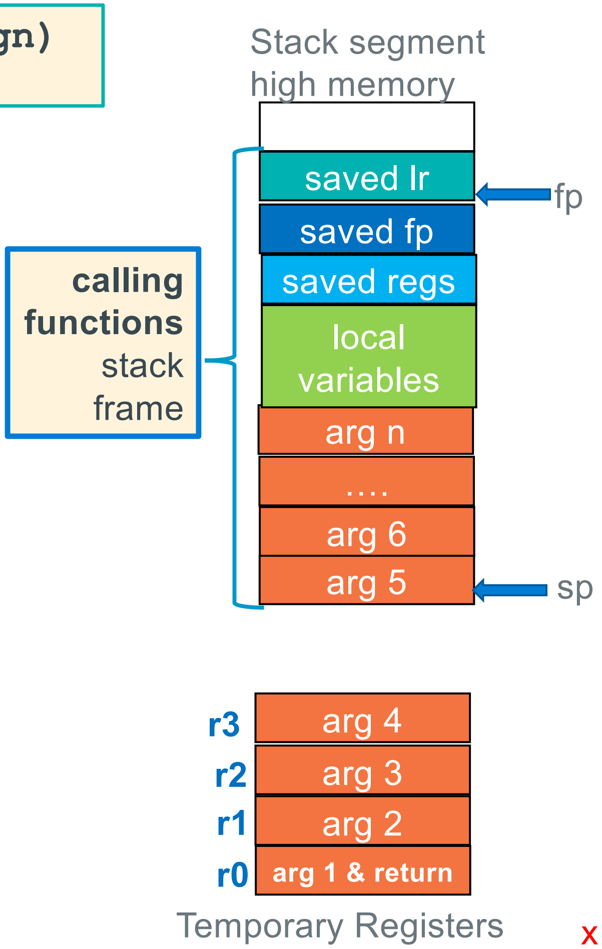
```
fillbuf:
    push    {fp, lr}         // stack frame
    add     fp, sp, FP_OFF   // set fp to base

    add     r1, r1, r0       // copy up to r1 = bufpt + cnt
    cmp     r0, r1           // are there any chars to fill?
    bge     .Ldone           // nope we are done

.Ldowhile:
    strb    r2, [r0]         // store the char in the buffer
    add     r0, 1            // point to next char
    cmp     r0, r1           // have we reached the end?
    blt     .Ldowhile        // if not continue to fill

.Ldone:
    sub     sp, fp, FP_OFF   // restore stack frame top
    pop     {fp, lr}         // restore registers
    bx      lr               // return to caller
```

x

# Passing More Than Four Arguments - 1

```
r0 = function(r0, r1, r2, r3, arg5, arg6, … argn)
              arg1, arg2, arg3, arg4, ...
```

- Each argument is a value that must fit in 32-bits
- **Args > 4 are in the caller's stack frame and arg 5 always starts at fp+4**
  - At the function call (bl) sp points at arg5
  - Additional args are higher up the stack, with one argument "slot" every 4-bytes
- Called functions have the right to change stack args just like they can change the register args!
- Caller must assume all args including ones on the stack are changed by the caller

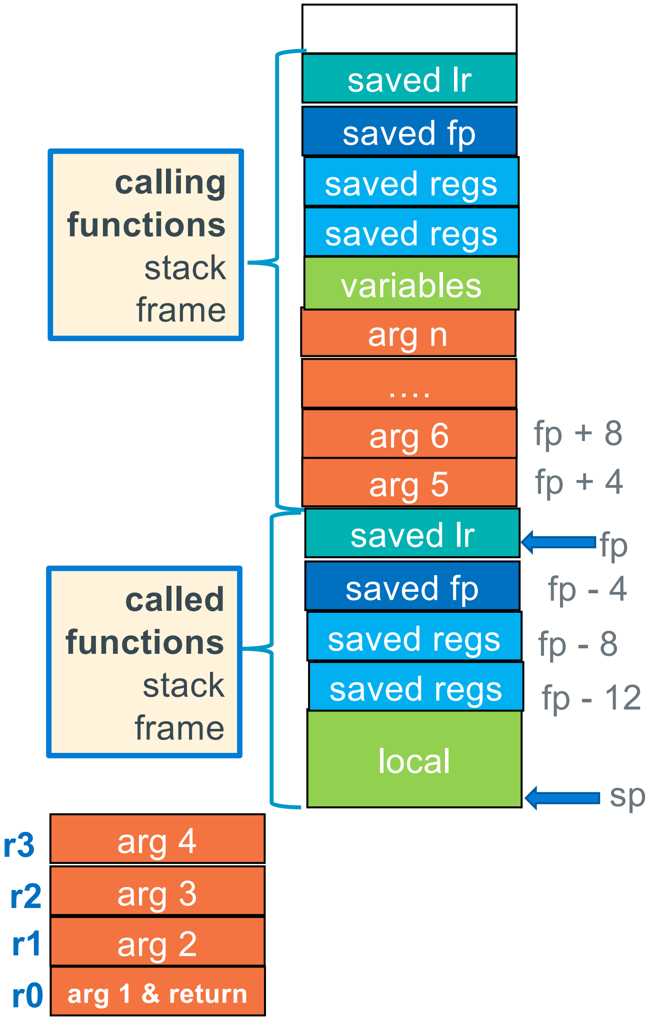


# Passing More Than Four Arguments - 2

```
r0 = function(r0, r1, r2, r3, arg5, arg6, … argn)
              arg1, arg2, arg3, arg4, ...
```

- **Addressing rules**
  - Adding to fp to get arg address in caller's frame
  - Subtracting from fp are addresses in called frame
- Why does it work this way?
- This "algorithm" for finding args was designed to enable languages to have variable argument count functions like:

```
printf("conversion list", arg0, … argn);
```



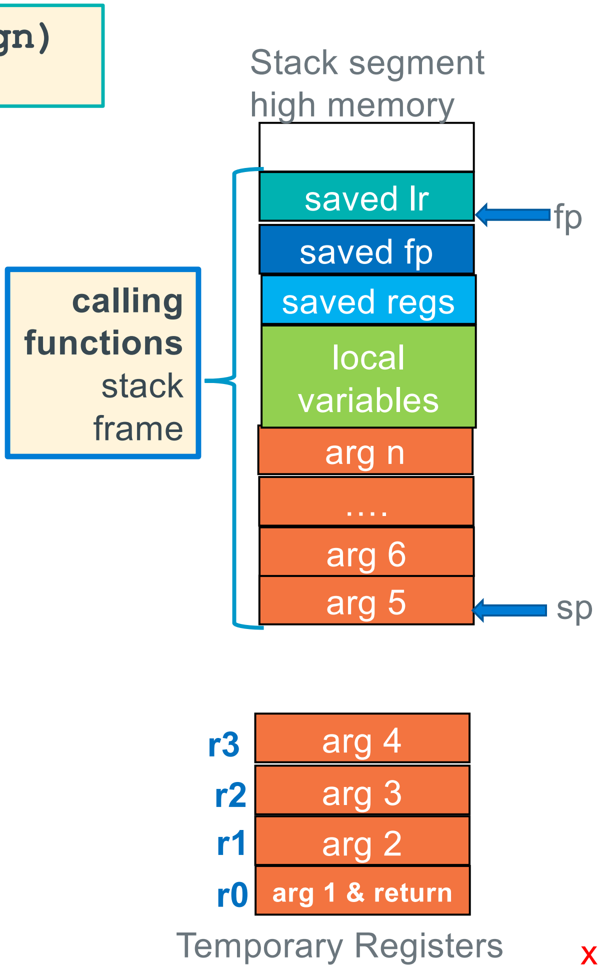
Temporary Registers

x

# Passing More Than Four Arguments – Calling Function

```
r0 = function(r0, r1, r2, r3, arg5, arg6, … argn)
              arg1, arg2, arg3, arg4, ...
```

- Calling function prior to making the call
  1. Evaluate first four args: place resulting values in r0-r3
  2. Arg 5 and greater are evaluated
  3. Store Arg 5 and greater parameter values on the stack
- **One arg value per slot**! – NO arrays across multiple slots
- chars, shorts and ints are directly stored
- Structs (not always), and arrays are passed via a pointer
- **Pointers** passed as output parameters usually contain an address ***that points at*** the stack, BSS, data, or heap



Version 1.09

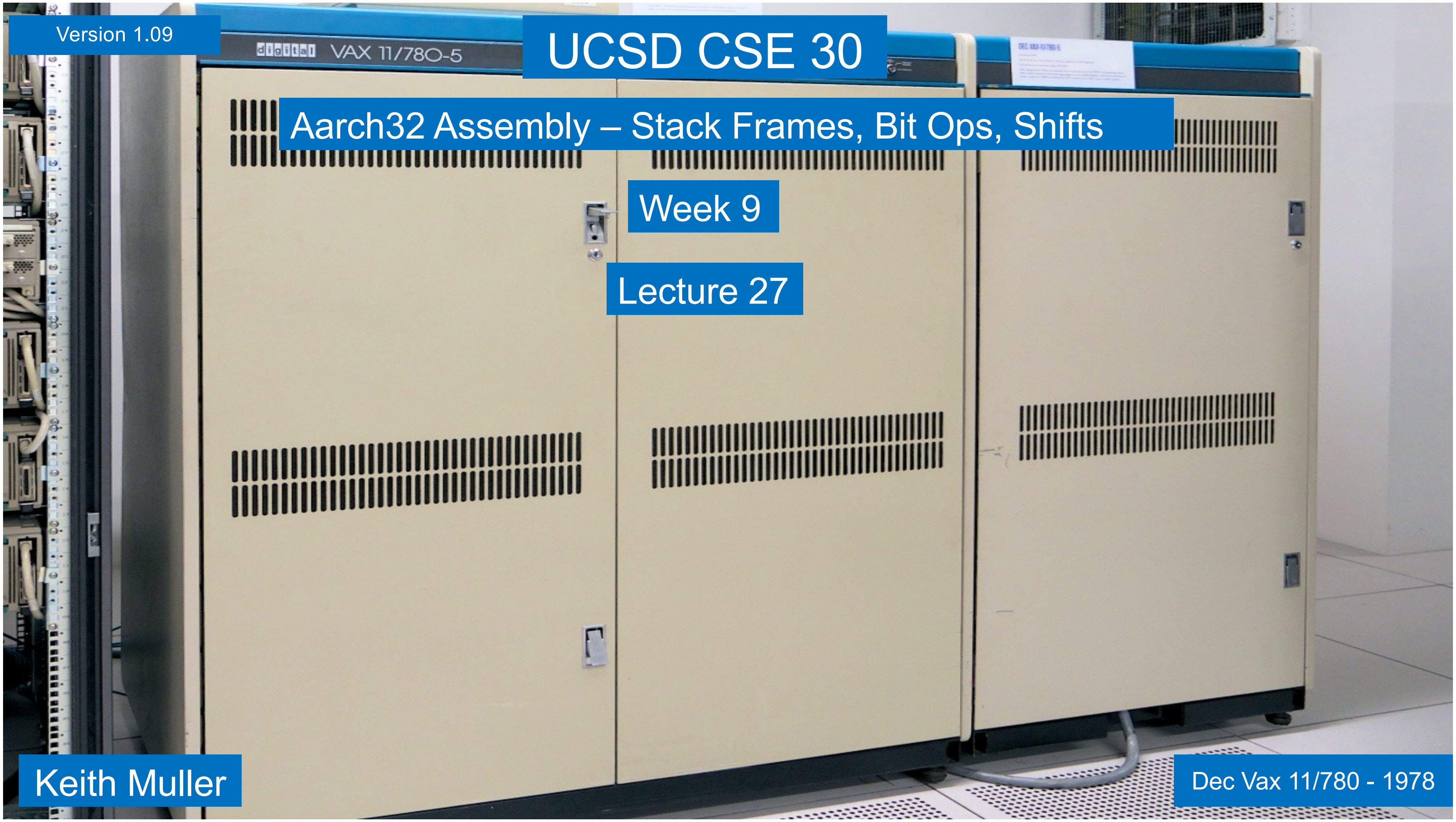
# UCSD CSE 30

## Aarch32 Assembly – Stack Frames, Bit Ops, Shifts

Week 9

Lecture 27

Keith Muller



Dec Vax 11/780 - 1978

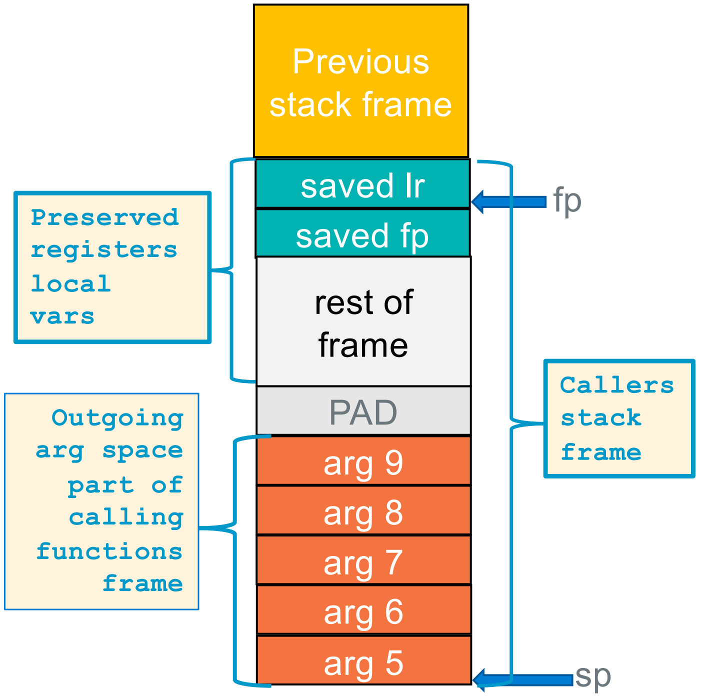
# Calling Function: Allocating Stack Parameter Space

At the point of a function call (and obviously at the start of the called function):

1. sp must point at arg5
2. arg5 **must be at an 8-byte boundary**,
   a) **padding** to force arg5 alignment is **placed above** the last **argument the called function is expecting**

**Approach**: Extend the stack frame to include enough space for stack arguments function with the greatest arg count

1. Examine every function call in the body of a function
2. Find the function call with greatest arg count, Determines space needed for outgoing args
3. Add the space needed to the frame layout



**Rules: At point of call**

1. **arg5 must be pointed at by sp**
2. **SP must be 8-byte aligned**

x

# Determining the Passed Parameter Area on The Stack

- Find the function called by main with the largest number of parameters
- That function determines the size of the Passed Parameter allocation on the stack

```
int main(void)
{
    /* code not shown */
    a(g, h);

   /* code not shown */
    sixsum(a1, a2, a3, a4, a5, a6);

   /* code not shown */

   b(q, w, e, r);
   /* code not shown */
}
```

largest arg count is 6
allocate space for 6 - 4 = 2 arg slots

# Passing More than Four Args – Six Arg Example

- Problem: Write and call a function that receives six integers and returns the sum
- First 4 parameters are in register r0 - r3 and the remaining argument are on the stack
- For this example, we will put all the locals on the stack

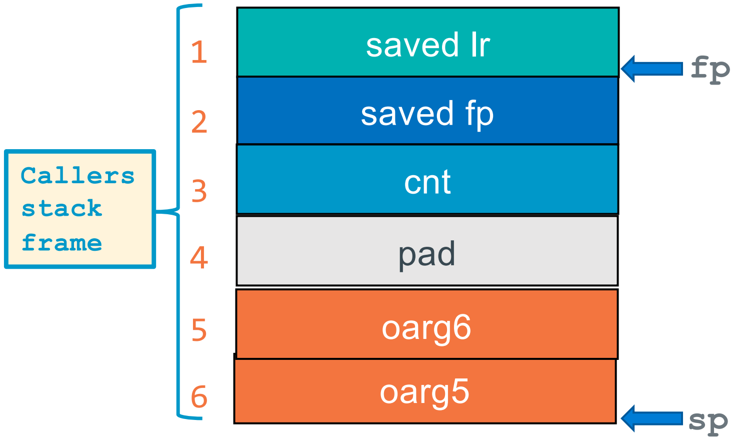
```
int main(void)
{
    int cnt = sixsum(1, 2, 3, 4, 5, 6);

    printf("the sum is %d\n", cnt);
    return EXIT_SUCCESS;
}
```

```
int
sixsum(int a1, int a2, int a3, int a4, int a5, int a6)
{
    return a1 + a2 + a3 + a4 + a5 + a6;
}
```

# Calling Function > 4 Args - 1

```
int cnt = sixsum(1, 2, 3, 4, 5, 6);
```

```
        .equ   FP_OFF,       4  // local base
           // NAME,          SIZE + prev_name
        .equ   CNT,          4 + FP_OFF
        .equ   PAD,          4 + CNT
        .equ   OARG6,        4 + PAD
        .equ   OARG5,        4 + OARG6
        .equ   FRAMESZ       OARG5 - FP_OFF
```



Callers stack frame

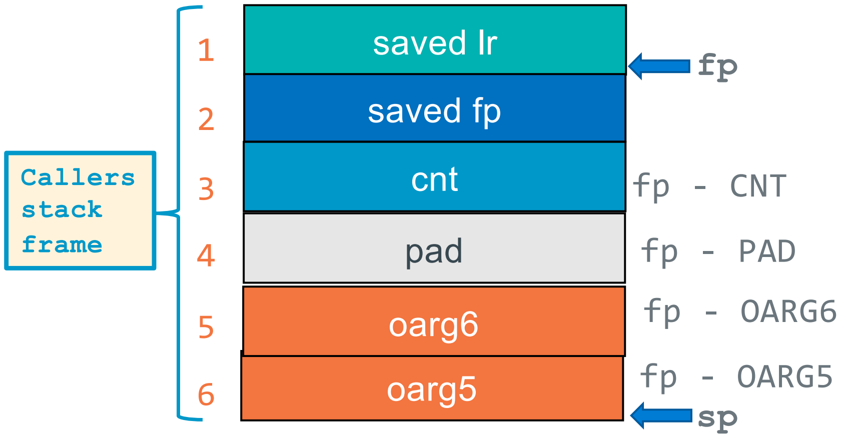
| | |
|---|---|
| 1 | saved lr |
| 2 | saved fp |
| 3 | cnt |
| 4 | pad |
| 5 | oarg6 |
| 6 | oarg5 |

x

# Calling Function > 4 Args - 2

```
int cnt = sixsum(1, 2, 3, 4, 5, 6);
```

```
.equ   FP_OFF,       4
.equ   CNT,          4 + FP_OFF
.equ   PAD,          4 + CNT
.equ   OARG6,        4 + PAD
.equ   OARG5,        4 + OARG6
.equ   FRAMESZ       OARG5 – FP_OFF
```



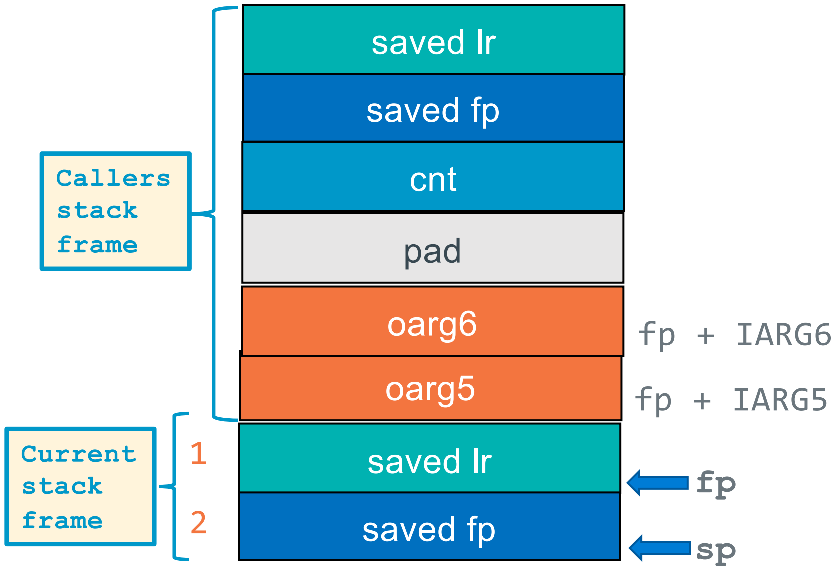
```
main:
    push    {fp, lr}
    add     fp, sp, FP_OFF
    sub     sp, sp, FRAMESZ

    mov     r0, 6
    str     r0, [fp, -OARG6]
    mov     r0, 5
    str     r0, [fp, -OARG5]
    mov     r3, 4
    mov     r2, 3
    mov     r1, 2
    mov     r0, 1
    bl      sixsum
    str     r0, [fp, -CNT]
    mov     r1, r0
    ldr     r0, =.Lpfstr
    bl      printf

    mov     r0, EXIT_SUCCESS
    sub     sp, fp, FP_OFF
    pop     {fp, lr}
    bx      lr
```

# Called Function > 4 Args

```
int sixsum(int a1, int a2, int a3, int a4, int a5, int a6)
    return a1 + a2 + a3 + a4 + a5 + a6;
```

```
.equ   IARG6,        8 // offset into caller's frame
.equ   IARG5,        4 // offset into caller's frame
.equ   FP_OFF,       4 // local base
```

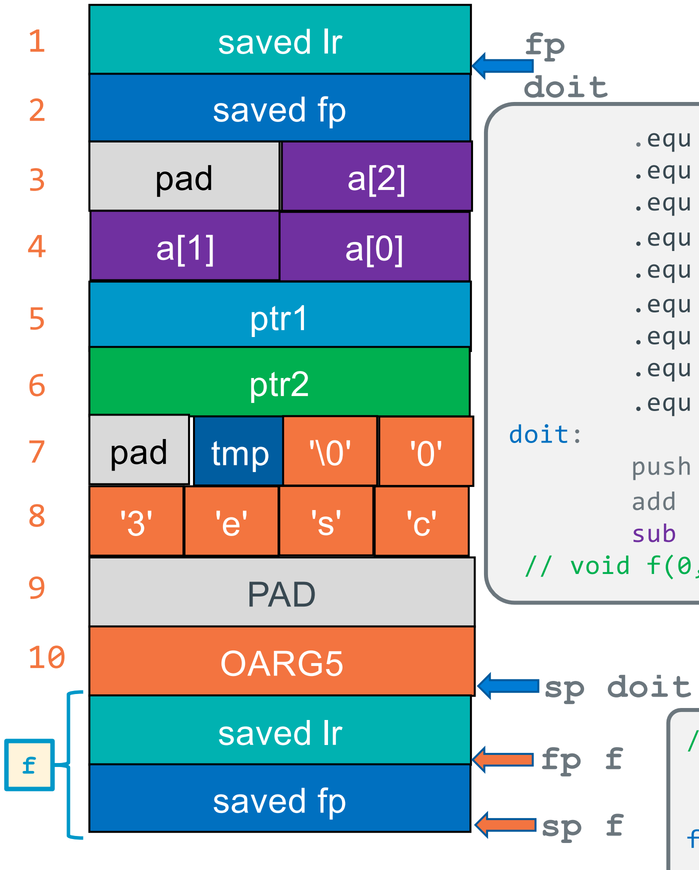
```
sixsum:
        push     {fp, lr}
        add      fp, sp, FP_OFF
        add      r0, r0, r1
        add      r0, r0, r2
        add      r0, r0, r3
        ldr      r1, [fp, IARG5]
        add      r0, r0, r1
        ldr      r1, [fp, IARG6]
        add      r0, r0, r1
        sub      sp, fp, FP_OFF
        pop      {fp, lr}
        bx       lr
```



x

# Recap: Passing pointers

Hint: Useful for PA5



```
        .equ      FP_OFF,   4
        .equ      A,        8 + FP_OFF
        .equ      PTR1,     4 + A
        .equ      PTR2,     4 + PTR1
        .equ      TMP,      2 + PTR2
        .equ      NM,       6 + TMP
        .equ      PAD,      4 + NM
        .equ      OARG5,    4 + PAD
        .equ      FRAMESZ   OARG5 - FP_OFF
doit:
        push     {fp, lr}
        add      fp, sp, FP_OFF
        sub      sp, sp, FRAMESZ
 // void f(0,1,2, &ptr1, &ptr2)
```

| | *How to pass* |
|---|---|
| &ptr1 | add  r3, fp, -PTR1 |
| &ptr2 | add  r4, fp, -PTR2<br>str  r4, [fp, -OARG5] |

Assume that while running, f() obtained two pointers from malloc() that it returns to doit in output arg4 and arg5

```
// void f(0,1,2, &ptr1, &ptr2)
        .equ      FP_OFF,   4
        .equ      IARG5,    4
f:
        push     {fp, lr}
        add      fp, sp, FP_OFF
```

| | *How to change* |
|---|---|
| &ptr1 | value to output is in r6<br>str r6, [r3] |
| &ptr2 | value to output is in r7<br>ldr r8, [fp, IARG5]<br>str  r7, [r8] |

x

# Bitwise (Bit to Bit) Operators in C

```
output = ~a;
```

| a | ~a |
|---|---|
| 0 | 1 |
| 1 | 0 |

```
output = a & b;
```

| a | b | a & b |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

& with 1 to let a **bit through**
& with 0 to **set a bit to 0**

```
output = a | b;
```

| a | b | a \| b |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

| with 1 to **set a bit to 1**
| with 0 to let a **bit through**

```
output = a ^ b; //EOR
```

| a | b | a ^ b |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

^ with 1 **will flip the bit**
^ with 0 to let a **bit through**

Bitwise NOT

```

~ 1100
  ----
  0011
```

Bitwise AND

```
  0110
& 1100
  ----
  0100
```

Bitwise OR

```
  0110
| 1100
  ----
  1110
```

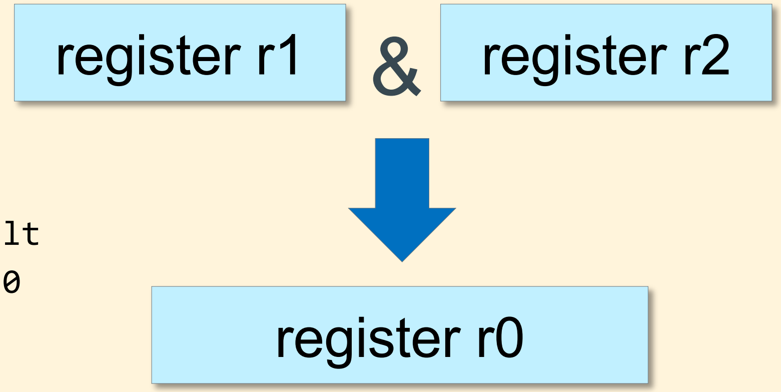
Bitwise EOR

```
  0110
^ 1100
  ----
  1010
```

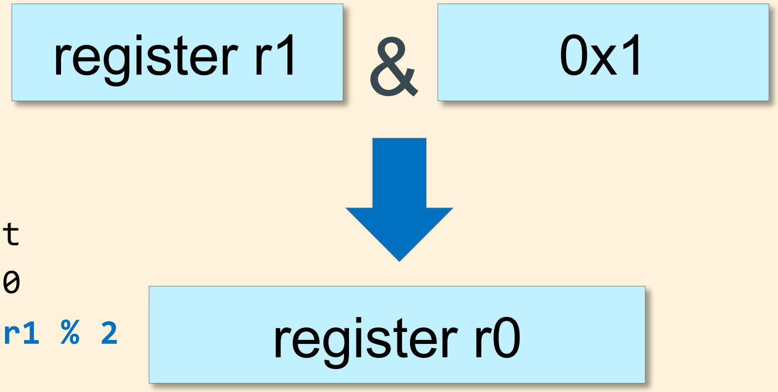
x

# First Look: AND Registers

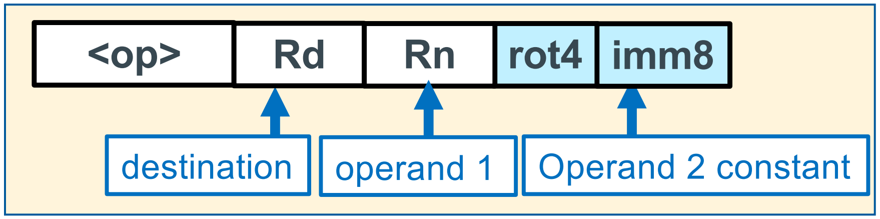
```
and   r0, r1, r2

// Copies all 32 bits
// of the bitwise result
// from r1 & r2 into r0
```

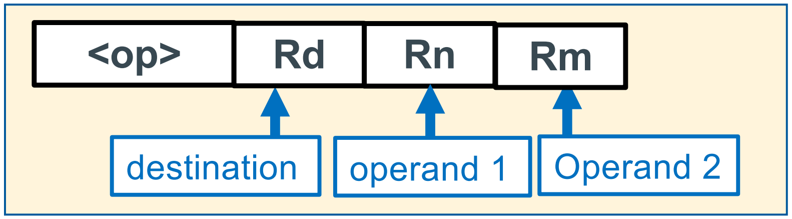


```
and   r0, r1, 1

// Copies all 32 bits
// of the bitwise result
// from r1 & 0x1 into r0
// Aside: This is r0 = r1 % 2
```



# Bitwise Instructions





```
<op>   Rd,   Rn, constant    // Rd = Rn <op> constant
<op>   Rd,   constant        // Rd = Rd <op> constant
<op>   Rd,   Rn,   Rm        // Rd = Rn <op> Rm
```

**Bytes**: 0 <= imm8 <= 255 + values from "rotating" rot 4 bits

| *Bitwise <op> description* | *<op> Syntax* | *Operation* |
|---|---|---|
| Bitwise **AND** | `and` $R_d$, $R_n$, Op2 | $R_d \leftarrow R_n$ & Op2 |
| **Bit Clear**<br>each bit in Op2 that is a 1, the same bit in $R_d$, is cleared | `bic` $R_d$, $R_n$, Op2 | $R_d \leftarrow R_n$ & ~Op2 |
| Bitwise **OR** | `orr` $R_d$, $R_n$, Op2 | $R_d \leftarrow R_n$ \| Op2 |
| Exclusive **OR** | `eor` $R_d$, $R_n$, Op2 | $R_d \leftarrow R_n$ ^ Op2 |

x

# Bit Masks: Masking - 1

- Bit masks access/modify specific bits in memory
- Masking act of applying a mask to a value
- `or:` 0 passes bit unchanged, 1 sets bit to 1
- `eor:` 0 passes bit unchanged, 1 inverts the bit
- `bic:` 0 passes bit unchanged, 1 clears it
- `and:` 0 clears the bit, 1 passes bit unchanged

mask force lower 16 bits to 1 "**mask on**" operation

```
orr  r1, r2, r3

DATA: r2 0xab ab ab 77

MASK: r3 0x00 00 ff ff lower half to 1
      ________________
RSLT: r1 0xab ab ff ff
```

mask to invert the lower 8-bits "**bit toggle**" operation

```
eor  r1, r2, r3

DATA: r2 0xab ab ab 77

MASK: r3 0x00 00 00 ff flip LSB bits
      ________________
RSLT: r1 0xab ab ab 88


MASK: r3 0x00 00 00 ff apply a 2nd time
      ________________
RSLT: r1 0xab ab ab 77 original value!
```

x

# Bit Masks: Masking - 2

mask to **extract top 8 bits** of r2 into r1

```
and   r1, r2, r3
DATA: r2 0xab ab ab 77
MASK: r3 0xff 00 00 00
RSLT: r1 0xab 00 00 00
```

mask to query the status of a bit "**bit status**" operation

```
and   r1, r2, r3
DATA: r2 0xab ab ab 77
MASK: r3 0x00 00 00 01 is bit 0 set?
RSLT: r1 0x00 00 00 01 (0 if not set)
```

mask to force lower 8 bits to 0 "**mask off**" operation

```
and   r1, r2, r3
DATA: r2 0xab ab ab 77
MASK: r3 0xff ff ff 00 clear LSB
RSLT: r1 0xab ab ab 00
```

clear  bit 5 to a 0 without changing the other bits

```
bic   r1, r2, r3
DATA: r2 0xab ab ab 77
MASK: r3 0x00 00 00 20 clear bit 5 (0010)
RSLT: r1 0xab ab ab 57
```

x

# Bit Masks: Masking - 3

mask to get **1's complement** operation

(like mvn)

```
eor  r1, r2, r3
DATA: r2 0xab ab ab 77
MASK: r3 0xff ff ff ff
RSLT: r1 0x54 54 54 88
```

**remainder (mod): num % d** where n ≥ 0 and d = $2^k$

mask = $2^k$ - 1 so for mod 2, mask = 2 -1 = 1

```
and  r1, r2, r3
DATA: r2 0xab ab ab 77
MASK: r3 0x00 00 00 01 (mod 2 even or odd)
RSLT: r1 0x00 00 00 01 (odd)
```

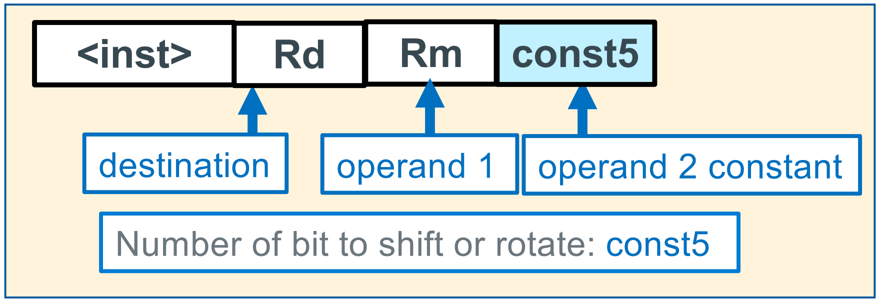
**remainder (mod): num % d** where n ≥ 0 and d = $2^k$

mask = $2^k$ -1 so for mod 16, mask = 16 -1 = 15

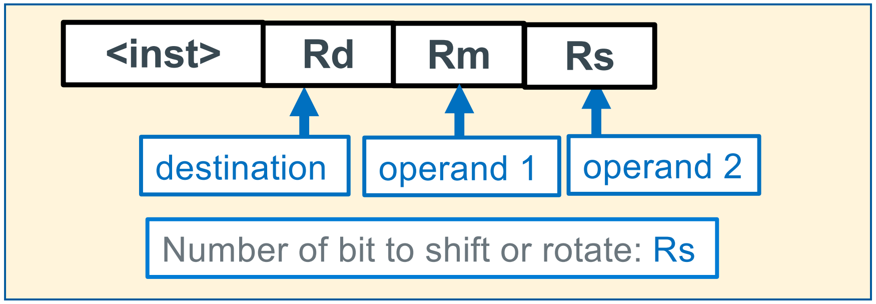
```
and  r1, r2, r3
DATA: r2 0xab ab ab 77
MASK: r3 0x00 00 00 0f (mod 16)
RSLT: r1 0xab 00 00 07 (if 0: divisible by)
```

x

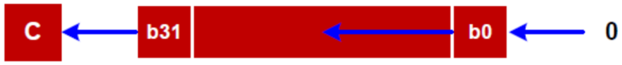
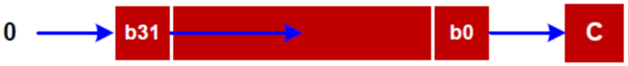
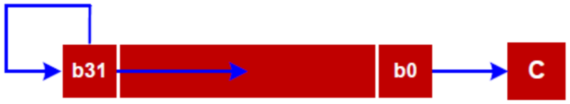
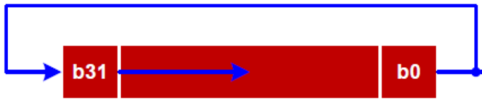
# Shift and Rotate Instructions



Number of bit to shift or rotate: const5



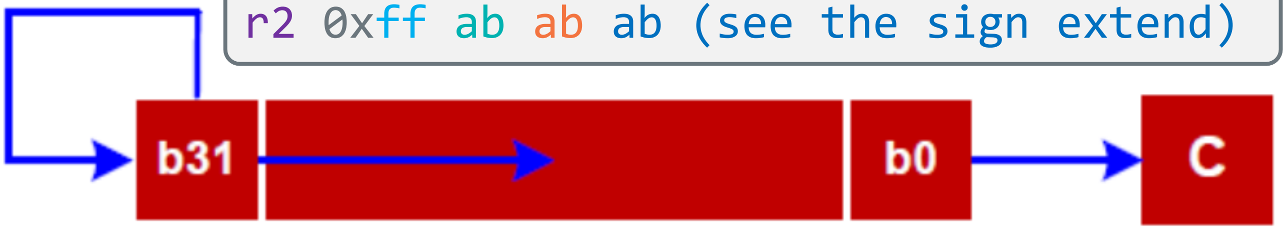
Number of bit to shift or rotate: Rs

| *Instruction* | *Syntax* | *Operation* | *Notes* | *Diagram* |
|---|---|---|---|---|
| Logical Shift Left | LSL $R_d$, $R_m$, *const5*<br>LSL $R_d$, $R_m$, $R_s$ | $R_d \leftarrow R_m$ << *const5*<br>$R_d \leftarrow R_m$ << $R_s$ | Zero fills<br>shift: 0 - 31 |  |
| Logical Shift Right | LSR $R_d$, $R_m$, *const5*<br>LSR $R_d$, $R_m$, $R_s$ | $R_d \leftarrow R_m$ >> *const5*<br>$R_d \leftarrow R_m$ >> $R_s$ | Zero fills<br>shift: 1 - 32 |  |
| Arithmetic Shift Right | ASR $R_d$, $R_m$, *const5*<br>ASR $R_d$, $R_m$, $R_s$ | $R_d \leftarrow R_m$ >> *const5*<br>$R_d \leftarrow R_m$ >> $R_s$ | Sign extends<br>shift: 1 - 32 |  |
| Rotate Right | ROR $R_d$, $R_m$, *const5*<br>ROR $R_d$, $R_m$, $R_s$ | $R_d \leftarrow R_m$ ror *const5*<br>$R_d \leftarrow R_m$ ror $R_s$ | right rotate<br>rot: 0 - 31 |  |

x

# Shift & Rotate Operations

```
asr r2, r0, 8

r0 0xab ab ab 77
r2 0xff ab ab ab (see the sign extend)
```



Test for sign
-1 if r0 negative

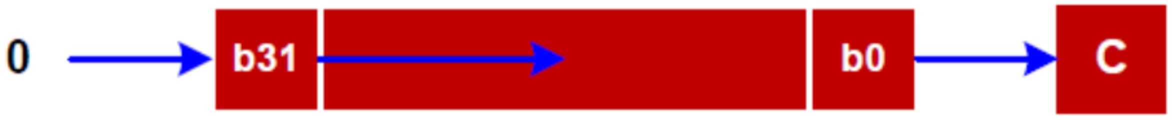
```
asr r2, r0, 31

r0 0xab ab ab 77
r2 0xff ff ff ff
```

Test for sign
0 if r0 positive

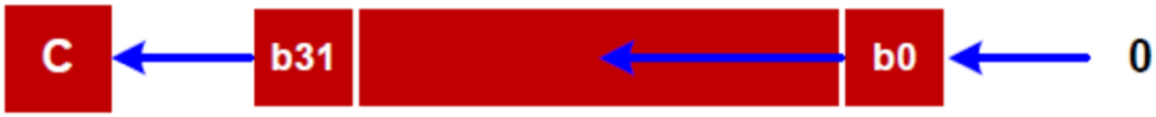
```
asr r2, r0, 31

r0 0x7b ab ab 77
r2 0x00 00 00 00
```

x

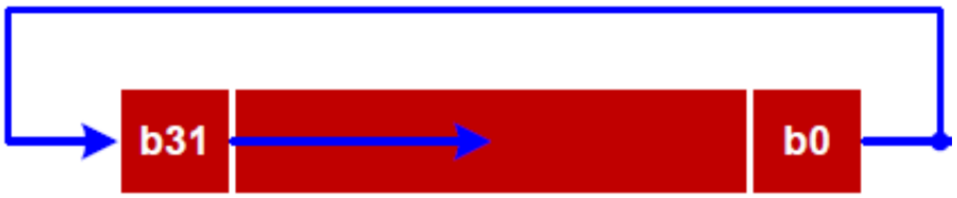
# Shift & Rotate Operations



```
lsr r2, r0, 8

r0 0xab ab ab 77
r2 0x00 ab ab ab
```



```
lsl r2, r0, 8

r0 0xab ab ab 77
r2 0xab ab 77 00
```



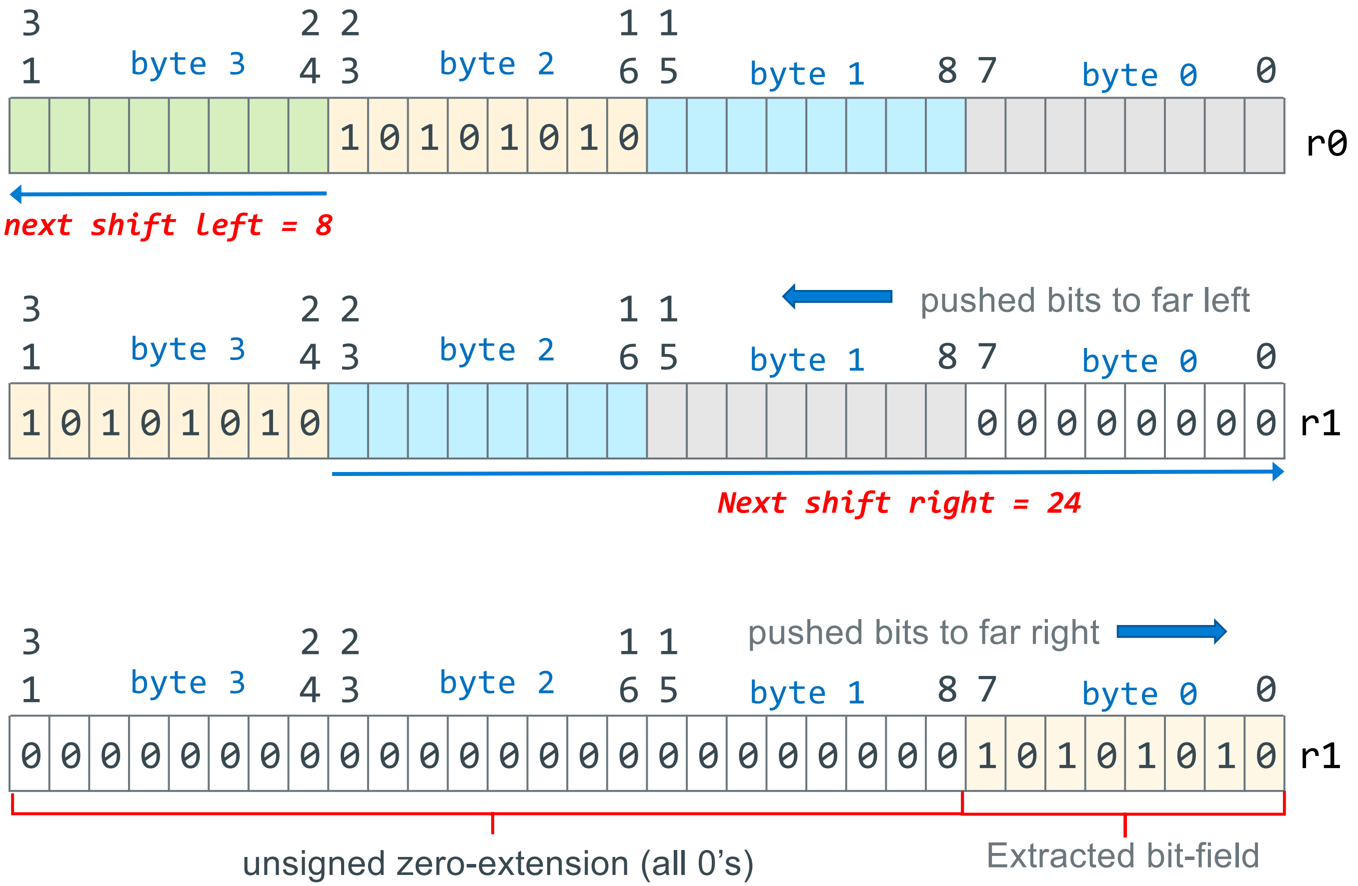
```
ror r2, r0, 8

r0 0xab ab ab 77
r2 0x77 ab ab ab
```

x

# Extracting Unsigned Bitfields

- Move byte 2 in r0 to byte 0 in r1

Hint: Useful for PA5

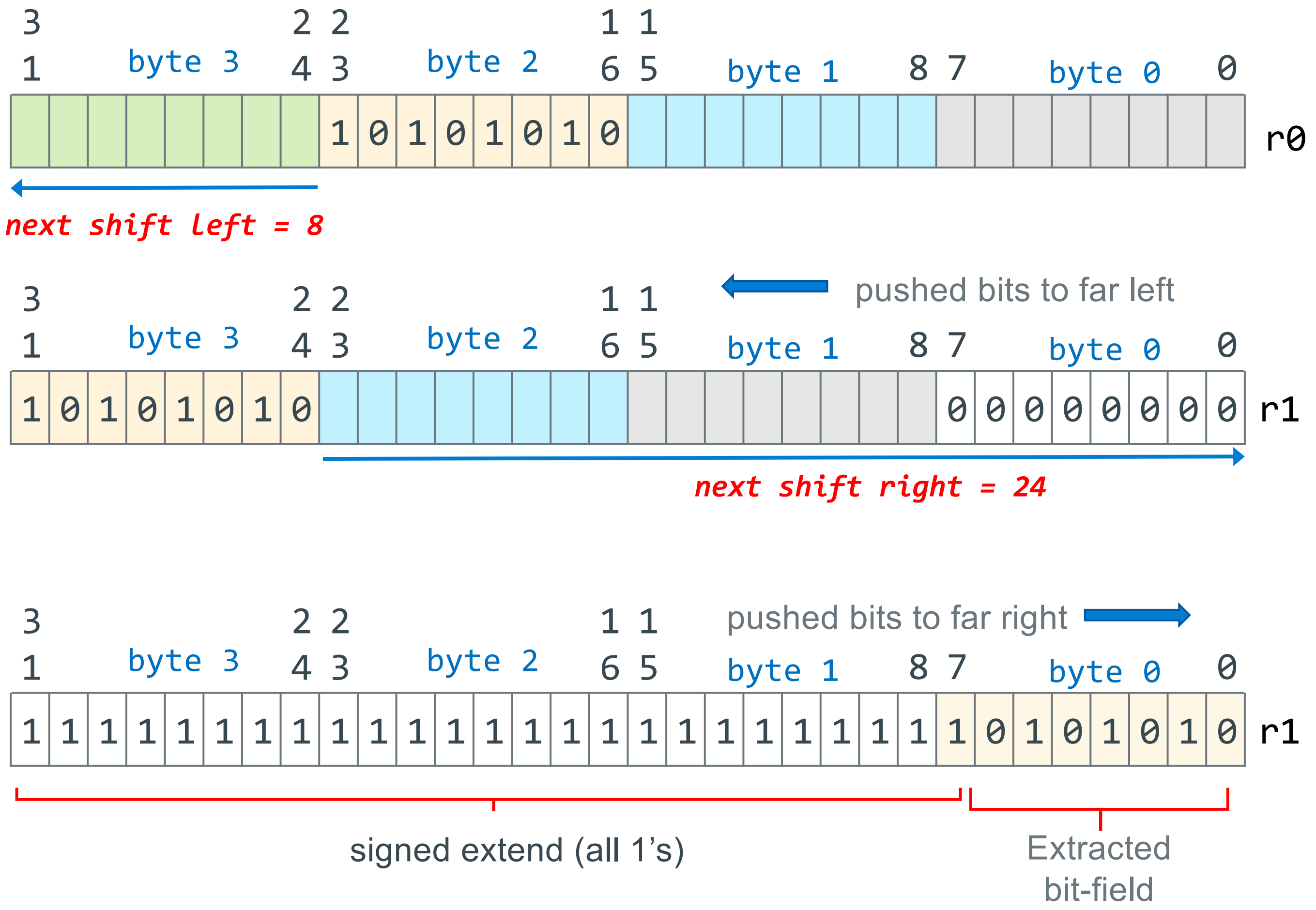


```
lsl  r1, r0, 8
lsr  r1, r1, 24
```

x

# Extracting Signed Bitfields

- Move byte 2 in r0 to byte 0 in r1



r0: `________ 10101010 ________ ________`

```
lsl   r1, r0, 8
```

r1: `10101010 ________ ________ 00000000`

```
asr   r1, r1, 24
```

r1: `11111111 11111111 11111111 10101010`

x

# Inserting Bitfields – Inserting Source Field into Destination Field

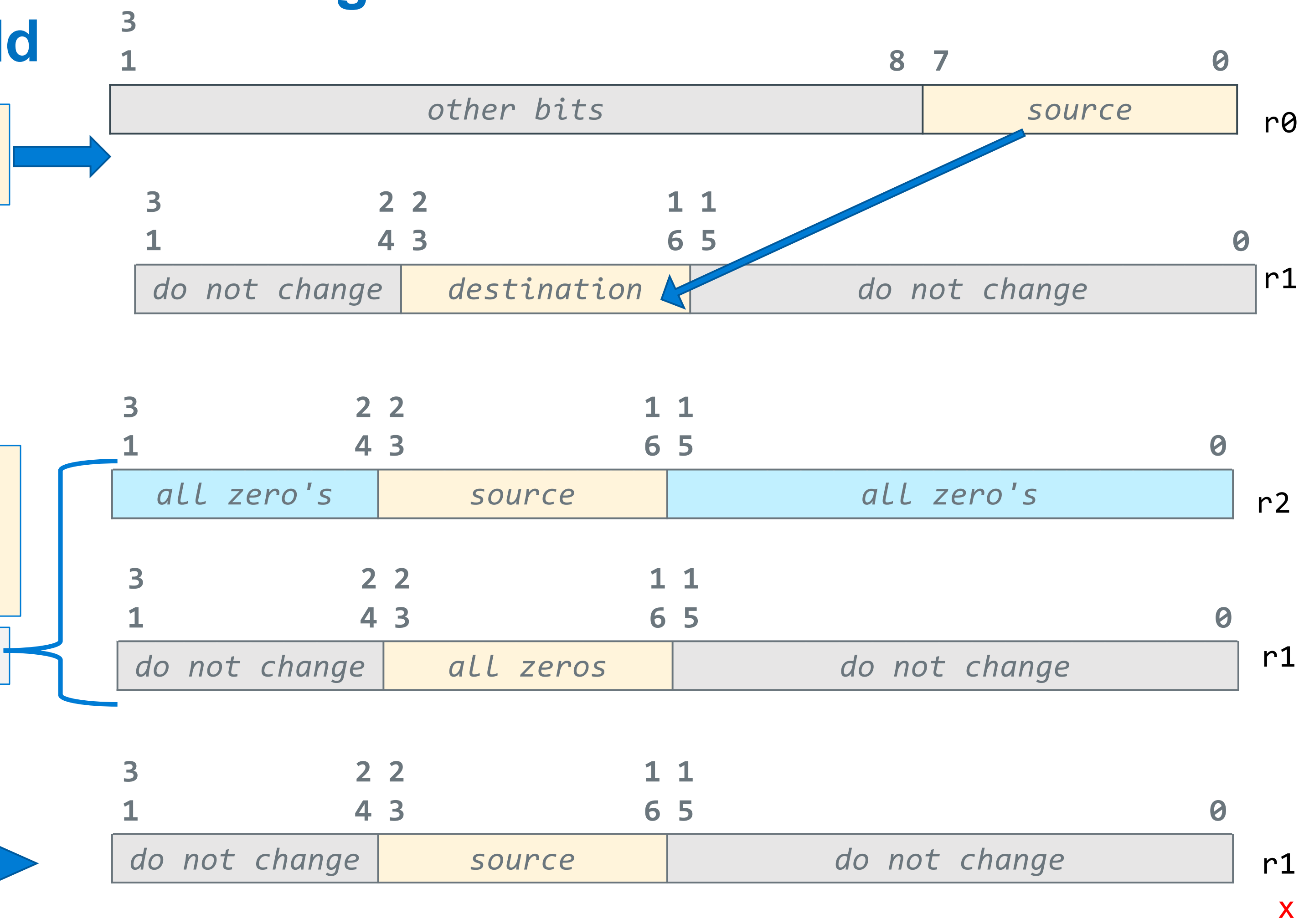
Task: Insert source into destination

| a | b | a \| b |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

Approach
(1) isolate source field
(2) clear destination field
(3) Bitwise or together

```
orr    r1, r1, r2
```

results in



# Creating a Mask

```
option #1 (1 mask)
ldr    r3, =0x00ffff0000


for a 0 mask
ldr    r3, =0xff0000ffff
```

| 3 | | | | | | | 2 | 2 | | | | | | | 1 | 1 | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | | | | | | | 4 | 3 | | | | | | | 6 | 5 | | | | | | | 8 | 7 | | | | | | | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

```
option #2 (1 mask)
small mask
mov    r3, 255




lsl    r3, r3, 16
  or do
ror    r3, r3, 16
```

| 3 | | | | | | | 2 | 2 | | | | | | | 1 | 1 | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | | | | | | | 4 | 3 | | | | | | | 6 | 5 | | | | | | | 8 | 7 | | | | | | | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

next shift left = 16 bits

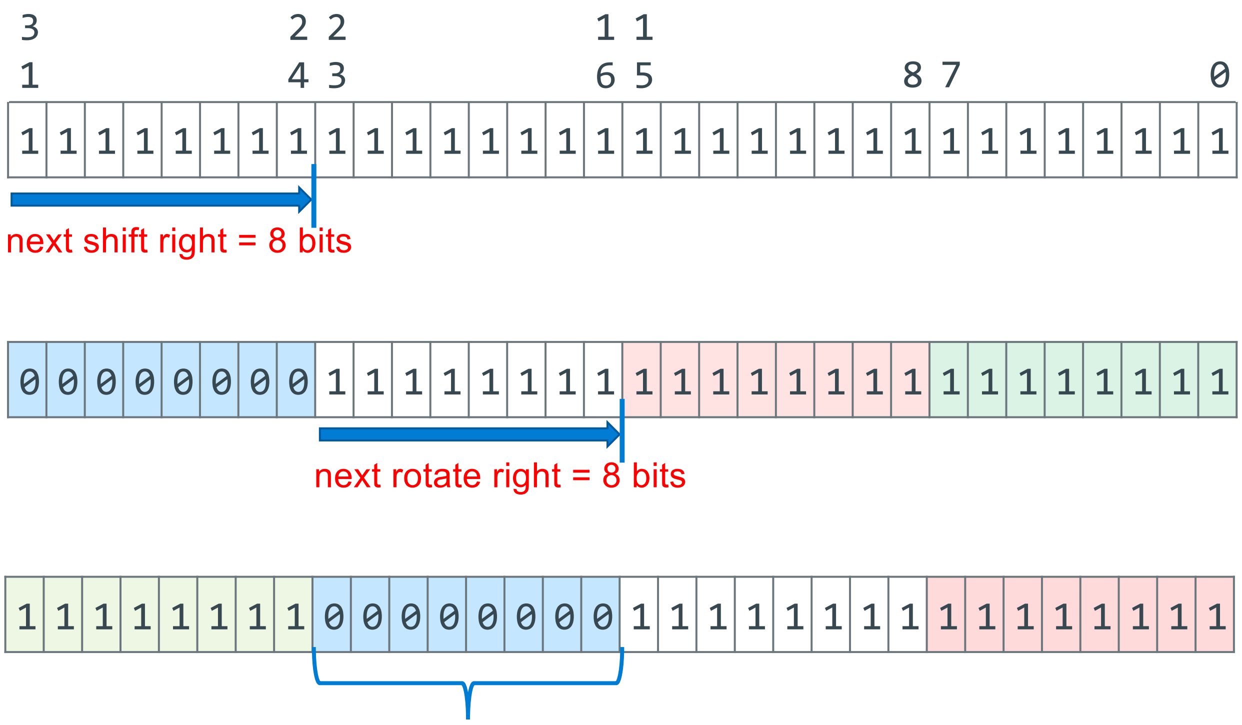
| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

x

# Creating a Mask- 0 mask

```
option #3
any size field

mov    r3, -1


asr    r3, r3, 8



ror    r3, r3, 8
```

number of bits you need in the mask, 8 for example



| 31 | | | | | | | 24 | 23 | | | | | | | 16 | 15 | | | | | | | 8 | 7 | | | | | | | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

next shift right = 8 bits

next rotate right = 8 bits

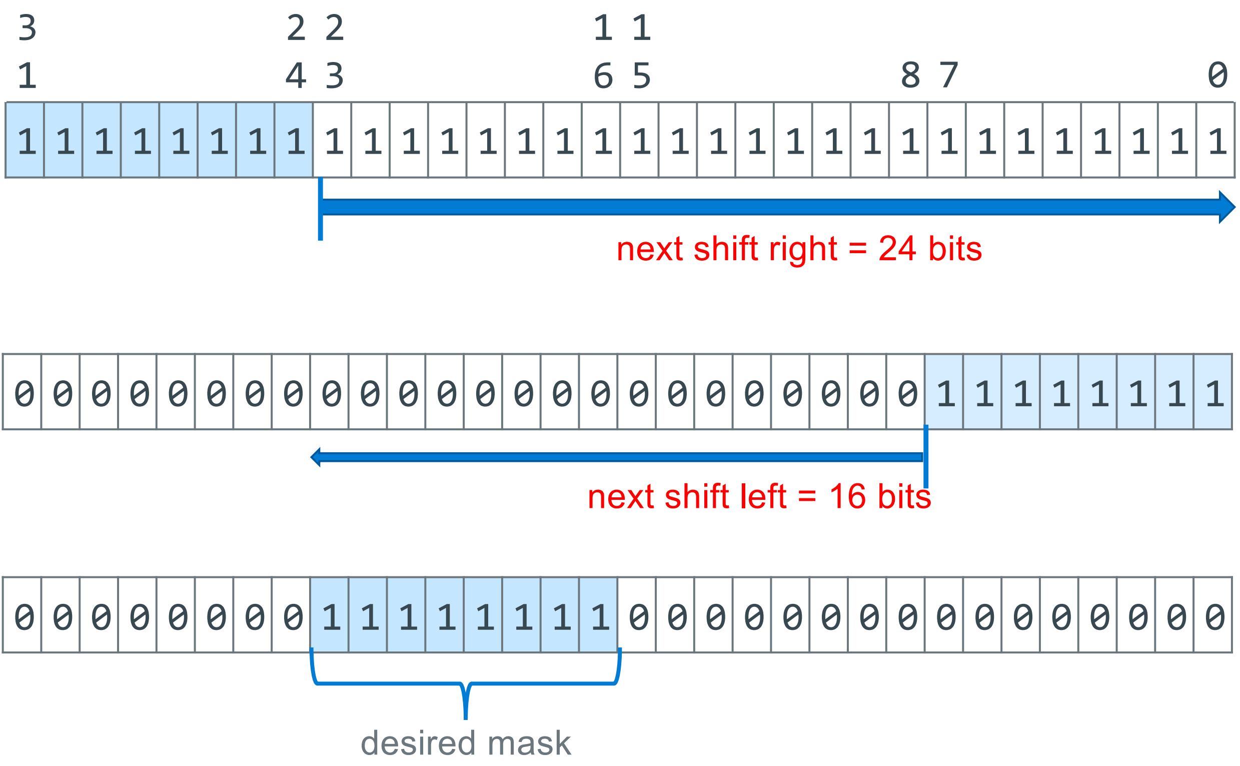
desired mask

x

# Creating a Mask- 1 mask

```
option #3
any size field

mov    r3, -1

lsr    r3, r3, 24

lsl    r3, r3, 16
```

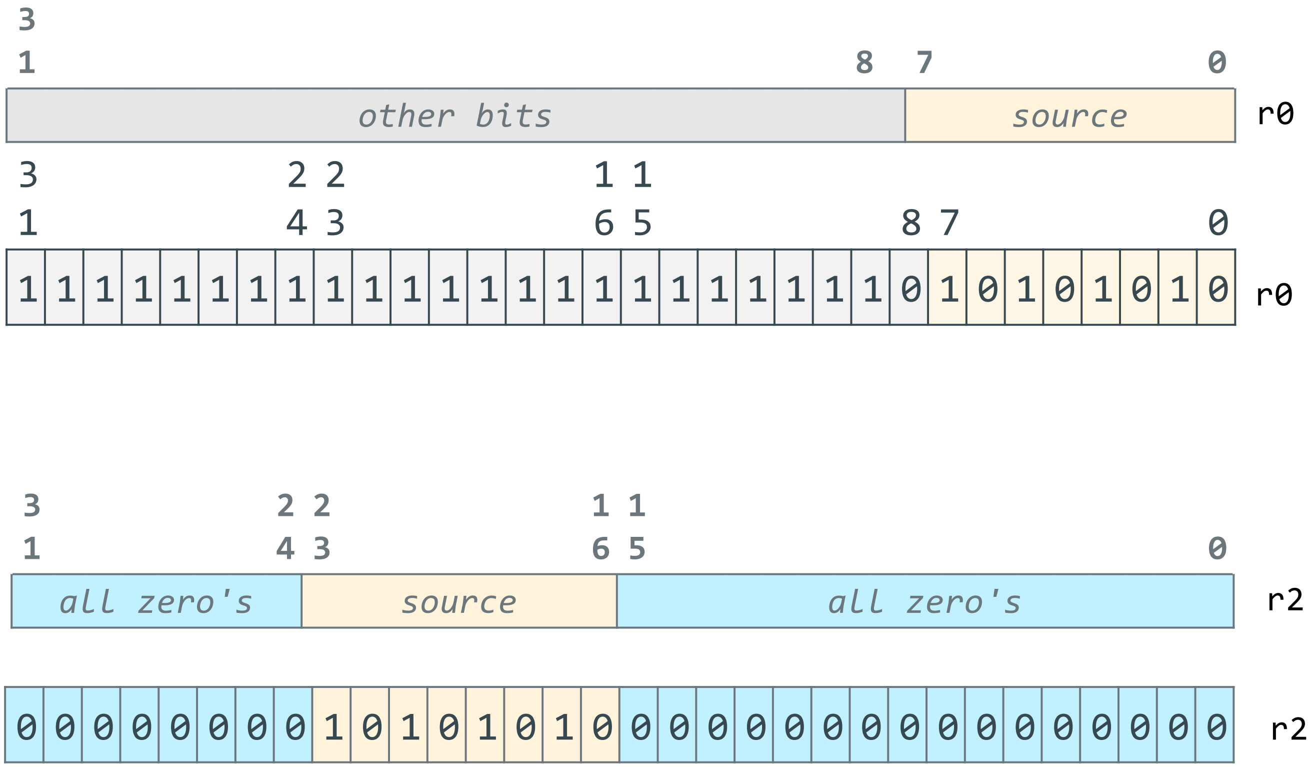
32 - number of bits you need in the mask, 8 for example is mask size



```
3                 2 2                 1 1                     
1                 4 3                 6 5                 8 7                 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1

0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

x

# Inserting Bitfields – Isolating the Source Field



r0: 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 0 1 0 1 0 1 0

r2: 0 0 0 0 0 0 0 0 1 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

```
isolate source field

lsl    r2, r0, 24
lsr    r2, r2, 8
```

x

# Inserting Bitfields – Clearing the Destination Field

| 31 | | | | | | | 24 | 23 | | | | | | | 16 | 15 | | | | | | | | | | | | | | | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *do not change* | | | | | | | | *destination* | | | | | | | | *do not change* | | | | | | | | | | | | | | | | r1 |
| 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | r1 |

```
create a 1 mask
```

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | r3 |

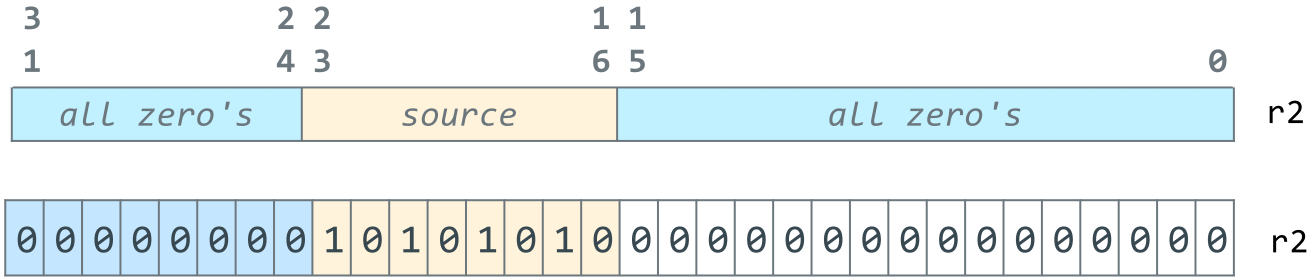
```
clear the
destination field

bic    r1, r1, r3
```

| 31 | | | | | | | 24 | 23 | | | | | | | 16 | 15 | | | | | | | | | | | | | | | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *do not change* | | | | | | | | *all zeros* | | | | | | | | *do not change* | | | | | | | | | | | | | | | | r1 |
| 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | r1 |

x

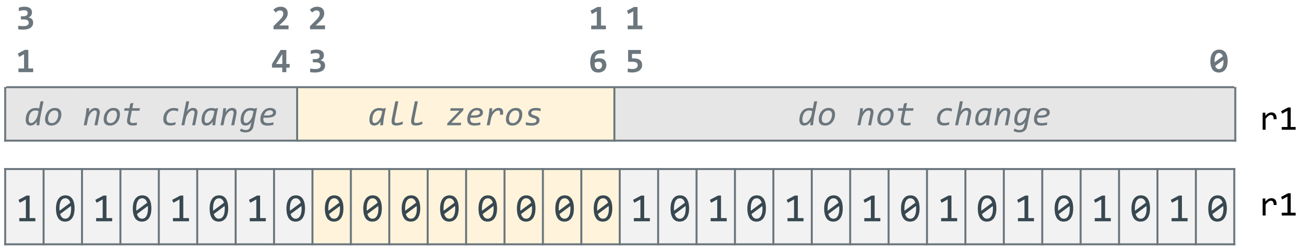
# Inserting Bitfields – Combining Isolated Source and Cleared Destination

isolated source



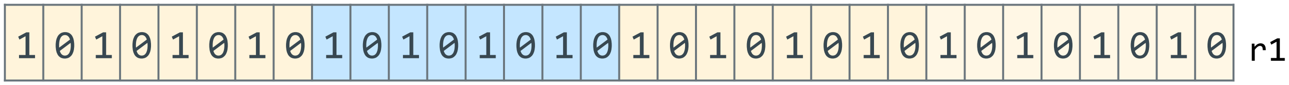
```
0 0 0 0 0 0 0 0 1 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0   r2
```

field cleared in destination



```
1 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0   r1
```

inserted field
orr r1, r1, r0

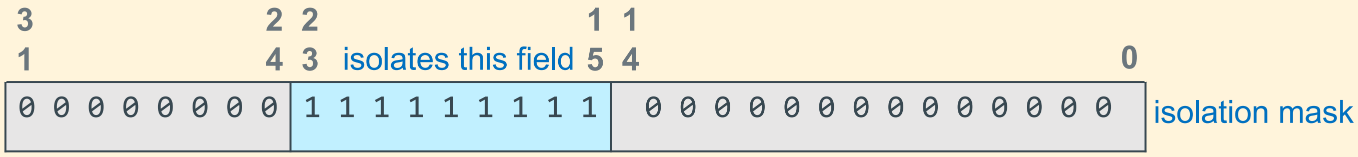


```
1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0   r1
```

x

# Masking Summary

**Isolate a field:** Use **`and`** with a mask of one's surrounded by zero's to select the bits that have a 1 in the mask, all other bits will be set to zero

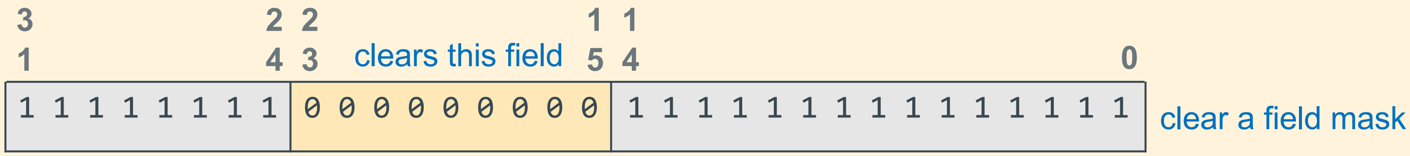


31 · 24 23 isolates this field 15 14 · 0

0 0 0 0 0 0 0 0 | 1 1 1 1 1 1 1 1 1 | 0 0 0 0 0 0 0 0 0 0 0 0 0 0

isolation mask

**Clear a field:** Use **`and`** with a mask of zero's surrounded by one's to select the bits that have a 1 in the mask, all other bits will be set to zero

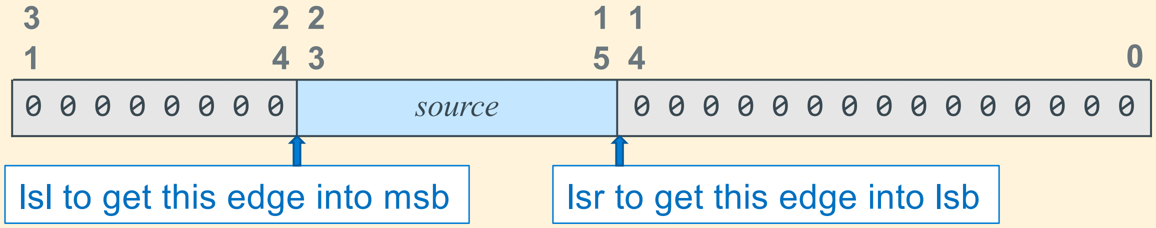


31 · 24 23 clears this field 15 14 · 0

1 1 1 1 1 1 1 1 | 0 0 0 0 0 0 0 0 0 | 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

clear a field mask

**Isolate a field:** Use **`lsr`** and **`lsl`** to get a field surrounded by zeros



31 · 24 23 · 15 14 · 0

0 0 0 0 0 0 0 0 | *source* | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

lsl to get this edge into msb

lsr to get this edge into lsb

# Reference For PA5: C Stream Functions Opening Files

```
FILE *fopen(char filename[], const char mode[]);
```

- Opens a stream to the specified file in specified file access mode
  - returns NULL on failure – always check the return value; make sure the open succeeded!
- Mode is a string that describes the actions that can be performed on the stream:

"r’’ Open for reading.

The stream is positioned at the beginning of the file. Fail if the file does not exist.

"w’’ Open for writing.

The stream is positioned at the beginning of the file. Create the file if it does not exist.

"a’’ Open for writing.

The stream is positioned at the end of the file. Create the file if it does not exist.

Subsequent writes to the file will always be at current end of file.

- An optional "+" following "r", "w", or "a" opens the file for both reading and writing

# Reference: C Stream Functions Closing Files and Usage

```
int fclose(FILE *stream);
```

- Closes the specified stream, forcing output to complete (eventually)
  - returns EOF on failure (often ignored as no easy recovery other than a message)
- Usage template for **`fopen()`** and **`fclose()`**
  1. Open a file with **`fopen()`** **`always`** checking the return value
  2. do i/o – keep calling stdio io routines
  3. close the file with **`fclose()`** when done with that I/O stream

# C Stream Functions Array/block read/write

- These do not process contents they simply **transfer** a fixed number of bytes to and from a buffer passed to them
- `size_t fwrite(void *ptr, size_t size, size_t count, FILE *stream);`
  - Writes an array of *count* ***elements*** of ***size*** bytes from **stream**
  - *Updates the write file pointer forward by the number of bytes written*
  - returns number of elements written
  - error is short element count or 0

- `size_t fread(void *ptr, size_t size, size_t count, FILE *stream);`
  - Reads an array of ***count elements*** of ***size*** bytes from *stream*
  - *Updates the read file pointer forward by the number of bytes read*
  - returns number of elements read, EOF is a return of 0
  - error is short element count or 0
- **I almost always set size to 1 to return bytes read/written**

x

# C fread/fwrite Example - 1

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#define BFSZ      8192 /* size of read */
int main(void)
{
  char fbuf[BFSZ];
  FILE *fin, *fout;
  size_t readlen;
  size_t bytes_copied = 0;
  retval = EXIT_SUCCESS;

  if (argc != 3){
    fprintf(stderr, "%s requires two args\n", argv[0]);
    return EXIT_FAILURE;
  }
  /* Open the input file for read */
  if ((fin = fopen(argv[1], "r")) == NULL) {
    fprintf(stderr,"fopen for read failed\n");
    return EXIT_FAILURE;
  }
  /*  Open the output file for write */
  if ((fout = fopen(argv[2], "w") == NULL) {
    fprintf(stderr, "fopen for write failed\n");
    fclose(fin);
    return EXIT_FAILURE;
  }
```

To handle bytes moved

```
% ls -ls ZZZ
ls: ZZZ: No such file or directory
% ./a.out cp.c ZZZ
bytes copied: 1122
% ls -ls cp.c ZZZ
8 -rw-r--r--  1 kmuller  staff  1122 Jul  2 08:51 ZZZ
8 -rw-r--r--  1 kmuller  staff  1122 Jul  2 08:49 cp.c
```

x

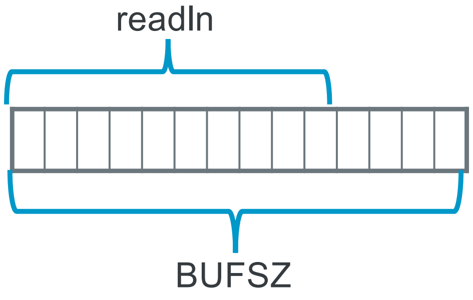
# C fread/fwrite Example - 2

```
  /* Read from the file, write to fout */

  while ((readlen = fread(fbuf, 1, BUFSIZ, fin)) > 0) {

    if (fwrite(fbuf, 1, readlen, fout) != readlen) {
       fprintf(stderr, "write failed\n");
        retval =  EXIT_FAILURE;
        break;
    }
    bytes_copied += readlen; //running sum bytes copied
  }

  if (retval == EXIT_FAILURE)
    printf("Failure Copy did not complete only ");
  printf("Bytes copied: %zu\n", bytes_copied);

  fclose(fin);
  fclose(fout);

  return retval;
}
```

By using an element size of 1 with a char buffer, this is byte I/O

Capture the bytes read so you know how many bytes to write

unless file length is an exact multiple of BUFSIZ, the last fread() will always be less than BUFSIZ which is why you write readln

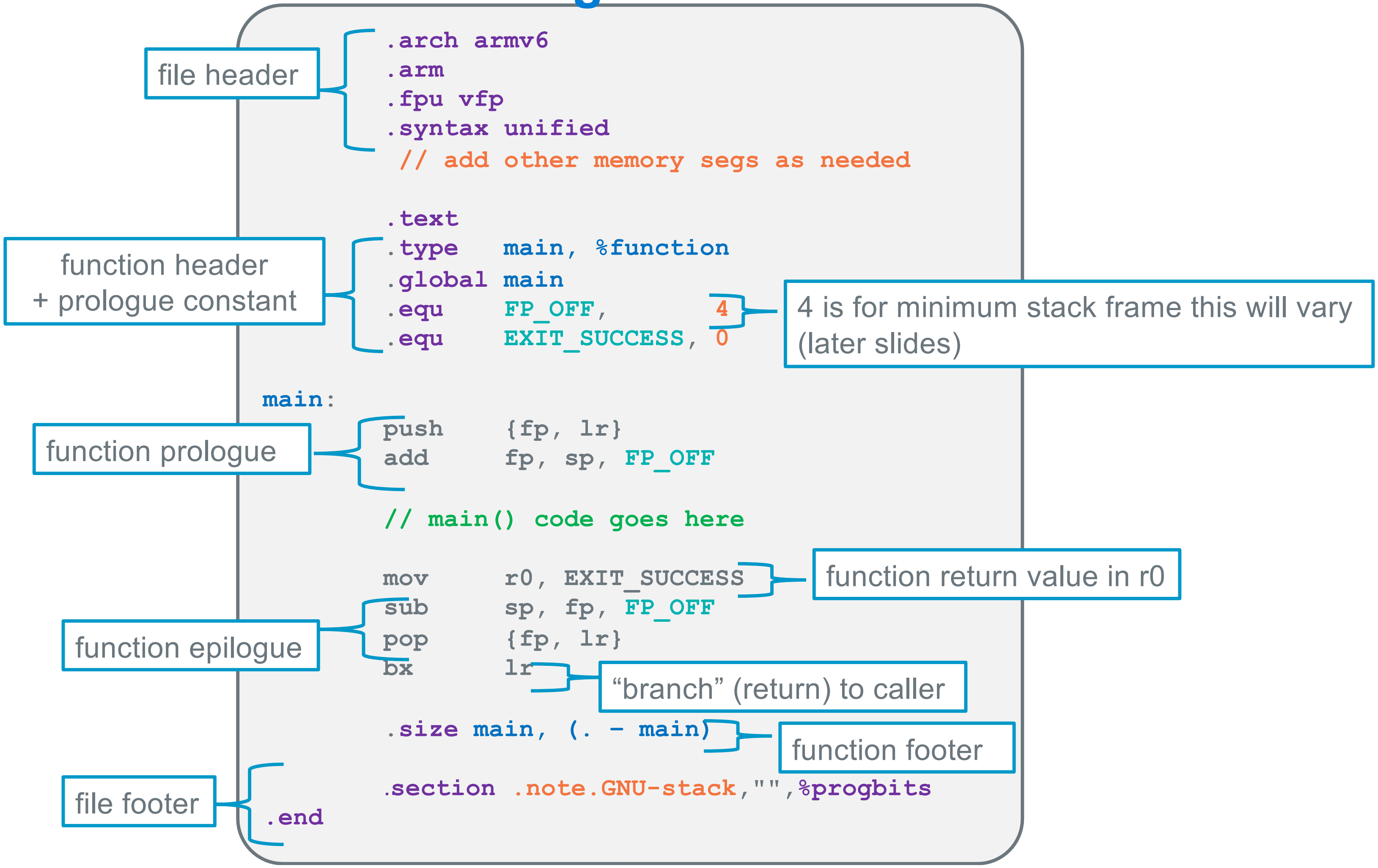


Jargon: the last record is often called the "runt"

x

# Extras

# main.S Source File Showing a minimum stack frame



```
        .arch armv6
        .arm
        .fpu vfp
        .syntax unified
         // add other memory segs as needed

        .text
        .type   main, %function
        .global main
        .equ    FP_OFF,         4
        .equ    EXIT_SUCCESS,   0

main:
        push    {fp, lr}
        add     fp, sp, FP_OFF

        // main() code goes here

        mov     r0, EXIT_SUCCESS
        sub     sp, fp, FP_OFF
        pop     {fp, lr}
        bx      lr

        .size main, (. - main)

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

file header
function header
+ prologue constant
4 is for minimum stack frame this will vary (later slides)
function prologue
function return value in r0
function epilogue
“branch” (return) to caller
function footer
file footer

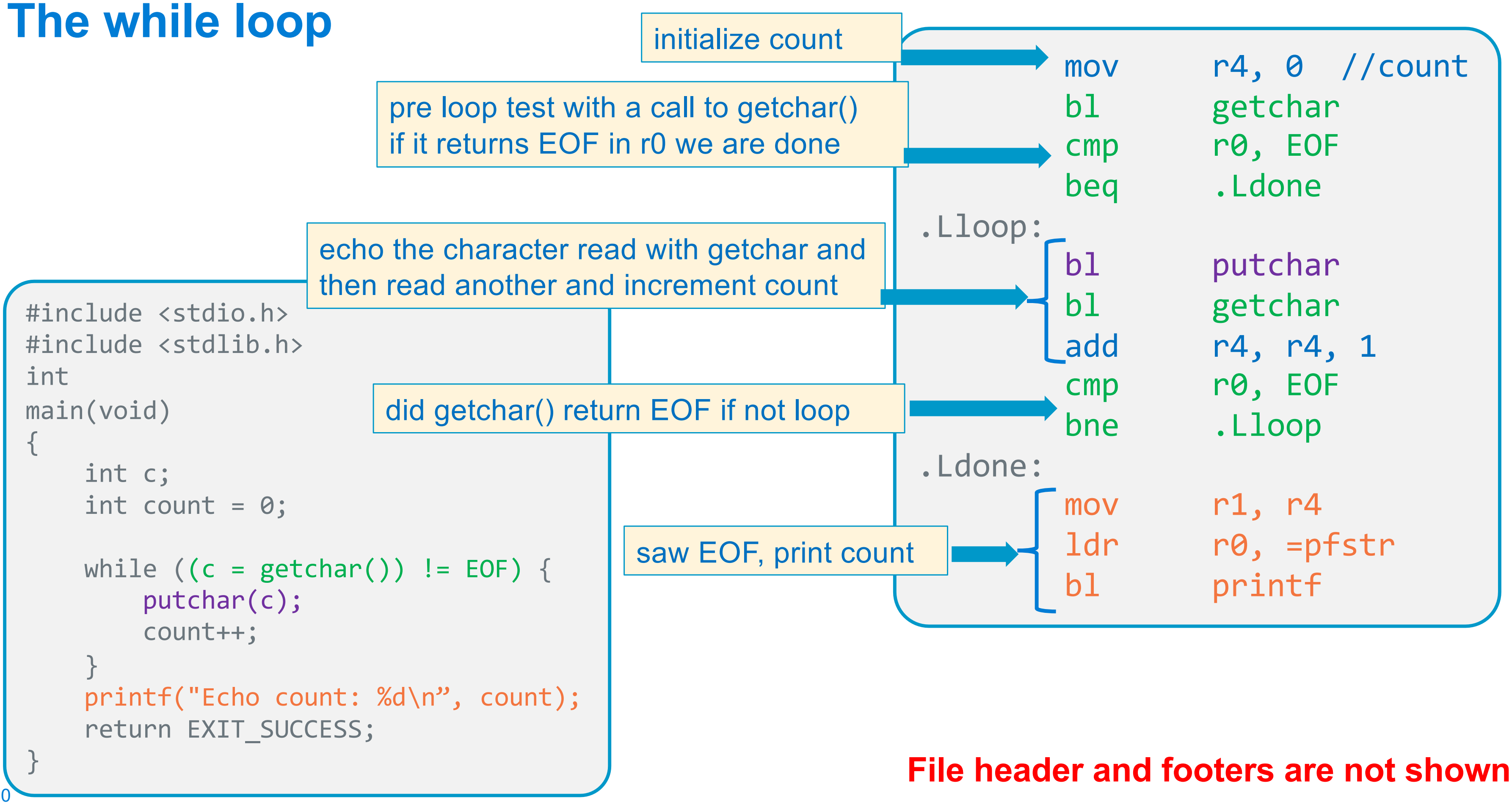
# putchar/getchar
# Setting up and Usage

```c
#include <stdio.h>
#include <stdlib.h>
int
main(void)
{
    int c;
    int count = 0;

    while ((c = getchar()) != EOF) {
        putchar(c);
        count++;
    }
    printf("Echo count: %d\n”, count);
    return EXIT_SUCCESS;
}
```

r0 r1

```
        .extern getchar
        .extern putchar
        .section .rodata
.Lfstr: .string  "Echo count: %d\n"
        .text
        .equ    EOF,          -1
        .type   main, %function
        .global main
        .equ    FP_OFF,    12
        .equ    EXIT_SUCCESS, 0
main:   push    {r4, r5, fp, lr}
        add     fp, sp, FP_OFF
        mov     r4, 0  //r4 = count

/* while loop code will go here */
.Ldone:
        mov     r1, r4 // count
        ldr     r0, =.Lfstr
        bl      printf
        mov     r0, EXIT_SUCCESS
        sub     sp, fp, FP_OFF
        pop     {r4, r5, fp, lr}
        bx      lr
        .size main, (. - main)
```

# Putchar/getchar: The while loop



```
#include <stdio.h>
#include <stdlib.h>
int
main(void)
{
    int c;
    int count = 0;

    while ((c = getchar()) != EOF) {
        putchar(c);
        count++;
    }
    printf("Echo count: %d\n”, count);
    return EXIT_SUCCESS;
}
```

initialize count

pre loop test with a call to getchar()
if it returns EOF in r0 we are done

echo the character read with getchar and then read another and increment count

did getchar() return EOF if not loop

saw EOF, print count

```
        mov     r4, 0  //count
        bl      getchar
        cmp     r0, EOF
        beq     .Ldone
.Lloop:
        bl      putchar
        bl      getchar
        add     r4, r4, 1
        cmp     r0, EOF
        bne     .Lloop
.Ldone:
        mov     r1, r4
        ldr     r0, =pfstr
        bl      printf
```

**File header and footers are not shown**

x

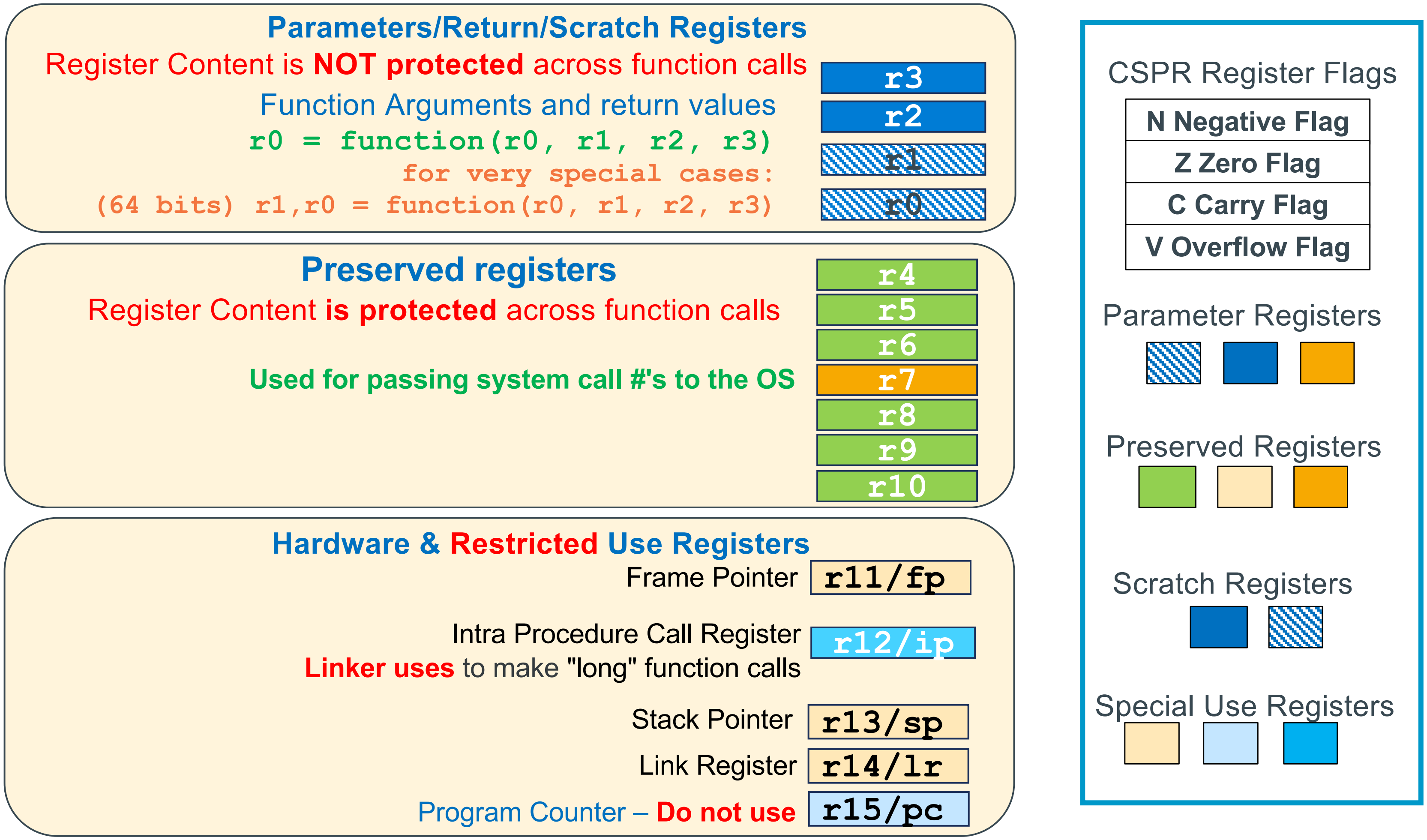
# printing error messages in assembly

```
.Lmsg0: .string "Read failed\n"
        ldr     r0, =.Lmsg0                  // read failed print error
        bl      errmsg
```

```
        // int errmsg(char *errormsg)
        // writes error messages to stderr
        .type   errmsg, %function                   // define to be a function
        .equ    FP_OFF,         4                   // fp offset in stack frame
errmsg:
        push    {fp, lr}                            // stack frame register save
        add     fp, sp, FP_OFF                      // set the frame pointer

        mov     r1, r0
        ldr     r0, =stderr
        ldr     r0, [r0]
        bl      fprintf
        mov     r0, EXIT_FAILURE                    // Set return value
        sub     sp, fp, FP_OFF                      // restore stack frame top
        pop     {fp, lr}                            // remove frame and restore
        bx      lr                                  // return to caller
        // function footer
        .size   errmsg, (. - errmsg)                // set size for function
```

x

# Reference: Registers and Flags – Programmers View



## Parameters/Return/Scratch Registers

Register Content is **NOT protected** across function calls

Function Arguments and return values

`r0 = function(r0, r1, r2, r3)`

`for very special cases:`

`(64 bits) r1,r0 = function(r0, r1, r2, r3)`

## Preserved registers

Register Content **is protected** across function calls

**Used for passing system call #'s to the OS**

## Hardware & Restricted Use Registers

Frame Pointer `r11/fp`

Intra Procedure Call Register `r12/ip`
**Linker uses** to make "long" function calls

Stack Pointer `r13/sp`

Link Register `r14/lr`

Program Counter – **Do not use** `r15/pc`

CSPR Register Flags

| |
|---|
| **N Negative Flag** |
| **Z Zero Flag** |
| **C Carry Flag** |
| **V Overflow Flag** |

Parameter Registers

▨ 🟦 🟧

Preserved Registers

🟩 🟨 🟧

Scratch Registers

🟦 ▨

Special Use Registers

🟨 🟦 🟦

x

# Bitwise versus C Boolean Operators

| *Meaning* | *Operator* | *Operator* | *Meaning* |
|---|---|---|---|
| Boolean AND | a && b | a & b | Bitwise AND |
| Boolean OR | a \|\| b | a \| b | Bitwise OR |
| Boolean NOT | !b | ~b | Biwise NOT |

Boolean operators **act on the entire value not the individual bits**

**& versus &&**

```
0x10 &   0x01 = 0x00 (bitwise)
0x10 &&  0x01 = 0x01 (Boolean)
```

**! versus ~**

```
~0x01 = 0xfffffffe (bitwise)
!0x01 = 0x0 (Boolean)
```

x