4. Instruction Set Architecture –
-branch instructions
-from C to Assembly

EECS 370 – Introduction to Computer Organization - Winter 2016

Profs. Valeria Bertacco & Reetu Das

EECS Department
University of Michigan in Ann Arbor, USA

Announcements

- Due on Thursday
  - Project 1.a
  - Homework 1

- Teaching on Thursday: Prof. Don Winsor

- Remember to report any exam conflict and need for special accommodations

Recap

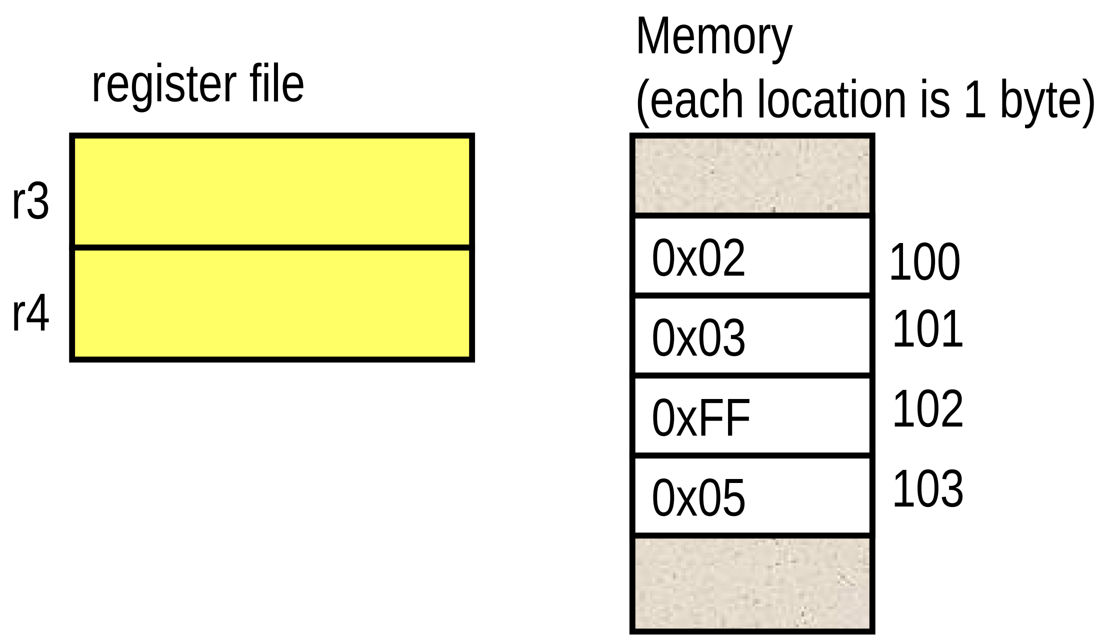
- LC-2K ISA
- ARM – arithmetic instructions
  - Special 2nd operand: reg / immediate / reg with shift/rot
- ARM – load/store instructions
  - Base+displacement
  - Base+register w. shift/rot
  - Pre-ndx/Post-ndx
  - Load/store word, halfword, byte – signed or unsigned
  - We use big-endian

ARM ISA MEM

Example Code Sequence

What is the final state of memory once you execute the following instruction sequence? (assume r0 stores the value 0)

```
ldr     r4, [r0, #100]
ldrsb   r3, [r0, #102]
str     r3, [r0, #100]
strb    r4, [r0, #102]
```



ARM ISA
MEM

Example Code Sequence – insn 1

What is the final state of memory once you execute the following instruction sequence? (assume r0 stores the value 0)

```
ldr     r4, [r0, #100]
ldrsb   r3, [r0, #102]
str     r3, [r0, #100]
strb    r4, [r0, #102]
```

register file

r3	
r4	0x0203 FF05

Memory
(each location is 1 byte)

0x02	100
0x03	101
0xFF	102
0x05	103

ARM ISA
MEM

Example Code Sequence – insn 2

What is the final state of memory once you execute the following instruction sequence? (assume r0 stores the value 0)

```
ldr     r4, [r0, #100]
ldrsb   r3, [r0, #102]
str     r3, [r0, #100]
strb    r4, [r0, #102]
```

register file

r3	0xFFFF FFFF
r4	0x0203 FF05

Memory
(each location is 1 byte)

0x02	100
0x03	101
0xFF	102
0x05	103

ARM ISA MEM

Example Code Sequence – insn 3

What is the final state of memory once you execute the following instruction sequence? (assume r0 stores the value 0)

```
ldr     r4, [r0, #100]
ldrsb   r3, [r0, #102]
str     r3, [r0, #100]
strb    r4, [r0, #102]
```

register file

r3	0xFFFF FFFF
r4	0x0203 FF05

Memory
(each location is 1 byte)

0xFF	100
0xFF	101
0xFF	102
0xFF	103

ARM ISA MEM

Example Code Sequence – insn 4

What is the final state of memory once you execute the following instruction sequence? (assume r0 stores the value 0)

```
ldr     r4, [r0, #100]
ldrsb   r3, [r0, #102]
str     r3, [r0, #100]
strb    r4, [r0, #102]
```

register file

r3	0xFFFF FFFF
r4	0x0203 FF05

Memory
(each location is 1 byte)

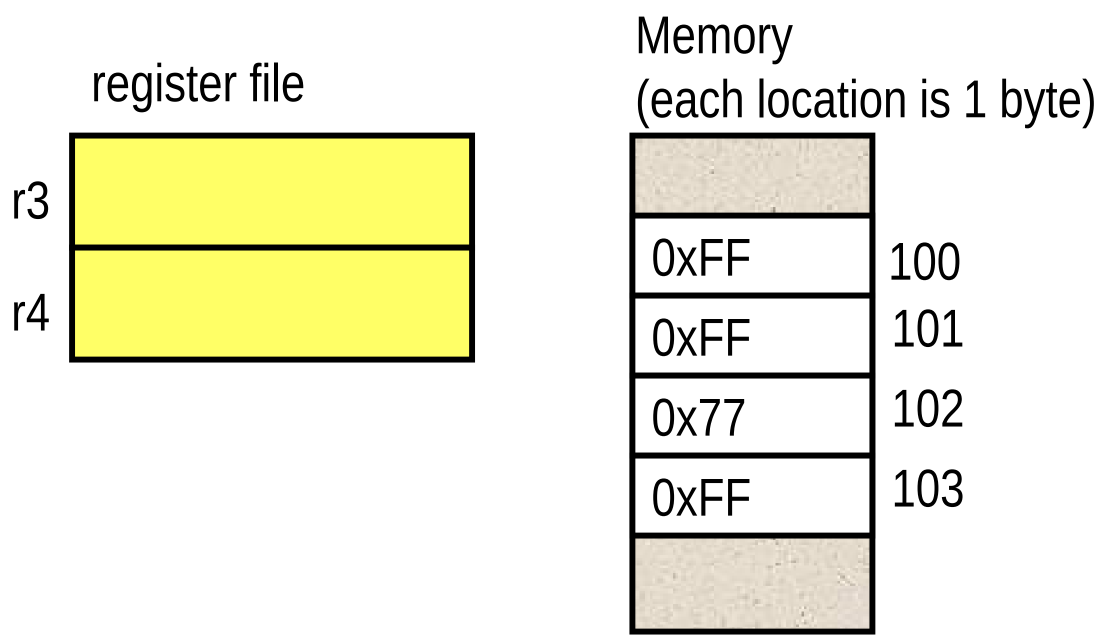
0xFF	100
0xFF	101
0x05	102
0xFF	103

Load/Store - Class Problem 3

ARM ISA MEM

What is the final state of memory once you execute the following instruction sequence? (assume r0 stores the value 0)

```
ldrh    r3, [r0, #100]
ldrb    r4, [r0, #102]
str     r3, [r0, #100]
strh    r4, [r0, #102]
```



ARM Sequencing Instructions

ARM ISA
BRANCH

- Sequencing instructions change the flow of instructions that are executed
  - This is achieved by modifying the program counter (r15)
- Conditional branches
  - If (condition_test) goto target_address
    - *condition_test* examines flags from the processor status word (PSR)
    - *target_address* is a 24-bit <u>word</u> displacement on current PC+8

> - cmp r1, r2
>   beq label
> - if (r1 == r2) then PC = label else PC = PC + 4

ARM Condition Codes Determine Branch Direction

ARM ISA BRANCH

- Most arithmetic/logic instructions can set condition codes in PSR
  - add, sub, cmp, and, eor, etc…
  - You need to add “S” to the instruction: adds, subs, ands,…
- Four primary condition codes evaluated:
  - N – set if the result is negative (i.e., bit 31 is non-zero)
  - Z – set if the result is zero (i.e., all 32 bits are zero)
  - C – set if last addition/subtraction had a carry/borrow out of bit 31
  - V – set if the last addition/subtraction produced an overflow (e.g., two negative numbers added together produced a positive result)
- Branch conditions:
  - eq – (Z == 1) gt – (Z == 0 && N == V) mi - ?
  - ne – (Z == 0) le – (Z == 1 || N != V) pl – ?
  - ge – (N == V) lt – (N != V) al – 1 (can use shorthand “b label”)

ARM ISA
BRANCH

Setting the Branch Displacement Field

- Target address is a 24-bit signed aligned displacement on current PC+8
  - Target = PC + 8 + 4 * 24_bit_signed_displacement
    - beq 1 // branch 3 instructions ahead if flag Z == 1
    - beq -3 // branch 1 instruction back if flag Z == 1
    - beq -2 // Infinite loop if flag Z == 1

Example code sequence

add
sub
mul
beq
ldr
PC+8 → str
mov

Other Branching Instructions

- cmp r2, r3 // branch to 4*offset+PC+8 if r2 $\neq$ r3
  bne offset
- cmp r2, r3 // branch to 4*offset+PC+8 if r2 < r3
  blt offset
- cmp r2, r3 // branch to 4*offset+PC+8 if r2 $\geq$ r3
  bge offset
- b offset // jump to 4*offset+PC+8
  // unconditional jump, is taken regardless of PSR flags
- mov r15, r3 // jump to address in r3 -- when is this useful?
- bl offset // put PC+4 into register r14 (LR) and jump to 4*offset+PC+8

Instruction Set Architecture (ISA) Design Lectures

- Lecture 2: Storage types and addressing modes
- Lecture 3 : LC-2K and ARM architecture
- Lecture 4 : Converting C to assembly – basic blocks
- Lecture 5 : Converting C to assembly – functions
- Lecture 6 : Translation software; libraries, memory layout

Converting C to Assembly

- Memory layout → memory addresses
- Control flow
- Procedure calls
- Expression trees
- Register allocation

Converting C to assembly – example 1

DATA LAYOUT

Write ARM assembly code for the following C expression:

C: a = b + names[i];

Assume that a is in r1, b is in r2, i is in r3, and the array names starts at address 1000 and <u>holds 32-bit integers</u>

```
mov     r5, r3, LSL #2          // calculate array offset
ldr     r4,  [r5, #1000]        // load names[i]
add     r1,  r2,  r4            // calculate b + names[i]
```

Converting C to assembly – example 2

DATA LAYOUT

Write ARM assembly code for the following C expression:

struct { int a; unsigned char b, c; } y; /* or a "class" in C++ */

y.a = y.b + y.c;

Assume that a pointer to y is in r1.

```
ldrb    r2,  [r1, #4]   // load y.b
ldrb    r3,  [r1, #5]   // load y.c
add     r4,  r2,  r3    // calculate y.b+y.c
str     r4,  [r1, #0]   // store y.a
```

How do you determine the offsets for the struct sub-fields?

Calculating Load/Store Addresses for Variables

DATA LAYOUT

```
short a[100];
char b;
int c;
double d;
short e;
struct {
      char f;
      int g[1];
      char h;
}  i;
```

Problem: Assume data memory starts at address 100, calculate the total amount of memory needed

a = 2 bytes * 100 = 200
b = 1 byte
c = 4 bytes
d = 8 bytes
e = 2 bytes
i = 1 + 4 + 1 = 6 bytes

total = 221, right or wrong?

Memory layout of variables

DATA LAYOUT

- For ARM, you cannot arbitrarily pack variables into memory
  - Need to worry about alignment
  - An N-byte variable must start at an address A, such that (A % N) == 0
  - Newer ARM processors will perform unaligned loads/stores, but they are VERY SLOW

- "Golden" rule – Address of a variable is aligned based on the size of the variable
  - **char** is byte aligned (any addr is fine)
  - **short** is half-word aligned (LSBit of addr must be 0)
  - **int** is word aligned (2 LSBit's of addr must be 0)

*Struct*ure/Class alignment

DATA LAYOUT

- Each field is laid out in the order it is declared using the Golden Rule for alignment

- Identify largest field
  - Starting address of overall struct is aligned based on the largest field
  - Size of overall struct is a multiple of the largest field
  - Reason for this is so we can have an array of structs

*Struct*ure Example

```
struct {
        char w;
        int x[3]
        char y;
        short z;
}
```

The largest field is **int** (4 bytes), hence:

- struct size is multiple of 4
- struct's starting addr is word aligned

Assume struct starts at location 1000,

char w → 1000

x[0] → 1004-1007, x[1] → 1008 – 1011, x[2] → 1012 – 1015

char y → 1016

short z → 1018 – 1019

Total size = 20 bytes!

Earlier Example – 2^{nd} Try

DATA LAYOUT

Assume data memory starts at address 100

```
short a[100];
char b;
int c;
double d;
short e;
struct {
      char f;
      int g[1];
      char h;
}  i;
```

→ 200 bytes → 100-299
→ 1 byte → 300-300
→ 4 bytes → 304-307
→ 8 bytes → 312-319
→ 2 bytes → 320-321
→ largest field is 4 bytes → start at 324
→ 1 byte → 324-324
→ 4 bytes → 328 - 331
→ 1 byte → 332-332
→ struct size is 12 bytes, spanning 324 – 335

236 bytes total!!

DATA LAYOUT

Class Problem 1

- How much memory is required for the following data, assuming that the data starts at address 200?

```
int a;
struct {double b, char c, int d} e;
char *f;
short g[20];
```

CTRL FLOW

If-Then-Else Example

Convert the following C code into ARM assembly (assume x is in r1, y in r2):

```
     int x, y;
     if (x == y)
         x++;
(L1) else
         y++;
(L2) …
```

Using Labels

```
    cmp  r1, r2
    bne  L1
    add r1, r1, #1
    b L2
L1: add r2, r2, #1
L2: …
```

Without Labels

```
cmp r1, r2
bne 1
add r1, r1, #1
b 0
add r2, r2, #1
```

Assemblers should deal with labels and assign displacements – why?

CTRL FLOW

Loop – Example

```
// assume all variables are integers
// i is in r1, start of a is at address 500, sum is in r2
for (i=0 ; i < 100 ; i++) {
      if (a[i] > 0) {
            sum += a[i];
      }
}
```

```
          mov   r1, #0
          mov   r4, #400
Loop1:    cmp   r1, r4
          bge   endLoop
          ldr   r5, [r1, #500]
          cmp   r5, #0
          ble   endIf
          add   r2, r2, r5
endIf:    add   r1, r1, #4
          b     Loop1
endLoop:
```

# of branch instructions
= 3*100 + 1 = 301

a.k.a. while-do template

CTRL FLOW

Same Loop, Different Assembly

```
// assume all variables are integers
// i is in r1, start of a is at address 500, sum is in r2
for (i=0 ; i < 100 ; i++) {
	if (a[i] > 0) {
		sum += a[i];
	}
}
```

```
         mov   r1, #0
         mov   r4, #400
Loop1:   ldr   r5, [r1, #500]
         cmp   r5, #0
         ble   endIf
         add   r2, r2, r5
endIf:   add   r1, r1, #4
         cmp   r1, r4
         blt   Loop1
endLoop:
```

# of branch instructions
= 2*100 = 200

a.k.a. do-while template

CTRL FLOW

Class Problem 2

Write the ARM assembly code to implement the following C code:

```
// assume ptr is in r1
// struct {int val; struct node *next;} node;
// struct node *ptr;

if ((ptr != NULL) && (ptr->val > 0))
    ptr->val++;
```

ARM conditional assembly

- All ARM instructions can be executed conditionally, not just branches
- Conditional assembly programs may boost performance because execution penalty is < branch misprediction

Example:

```
while (r0 != r1) {
        if (r0>r1) r0 = r0 - r1;
        else r1 =r1 – r0;
}
```

standard assembly

```
TOP     cmp r0, r1
        beq END
        blt LESS
        sub r0, r0, r1
        b TOP
LESS    sub r1, r1, r0
        b TOP
END ….
```

conditional assembly

```
TOP     cmp r0, r1
        subgt r0, r0, r1
        sublt r1, r1, r0
        bne TOP
        …
```

Instruction Set Architecture (ISA) Design Lectures

- Lecture 2: Storage types and addressing modes
- Lecture 3 : LC-2K and ARM architecture
- Lecture 4 : Converting C to assembly – basic blocks
- Lecture 5 : Converting C to assembly – functions
- Lecture 6 : Translation software; libraries, memory layout

Converting function calls to assembly code

C: printf(“hello world\n”);

- Need to pass parameters to the called function (printf)
- Need to save return address of caller
- Need to save register values
- Need to jump to printf

Execute instructions for printf()
Jump to to return address

- Need to get return value (if used)
- Restore register values

FUNCTION CALLS

Task 1: Passing parameters

- Where should you put all of the parameters?
  - Registers?
    - Fast access but few in number and wrong size for some objects
  - Memory?
    - Good general solution but where?

- ARM answer:
  - Registers and memory
    - Put the first few parameters in registers (if they fit) (r0 – r3)
    - Put the rest in memory on the call stack
  - Example:
    mov   r0, #1000 // put address of char array “hello world” in r0

Call stack

- ARM conventions (and most other processors) allocate a region of memory for the call stack
  - This memory is used to manage all the storage requirements to simulate function call semantics
    - Parameters (that were not passed through registers)
    - Local variables
    - Temporary storage (when you run out of registers and need somewhere to save a value)
    - Return address
    - Etc.
- Sections of memory on the call stack [a.k.a. **stack frames**] are allocated when you make a function call, and de-allocated when you return from a function.

FUNCTION CALLS

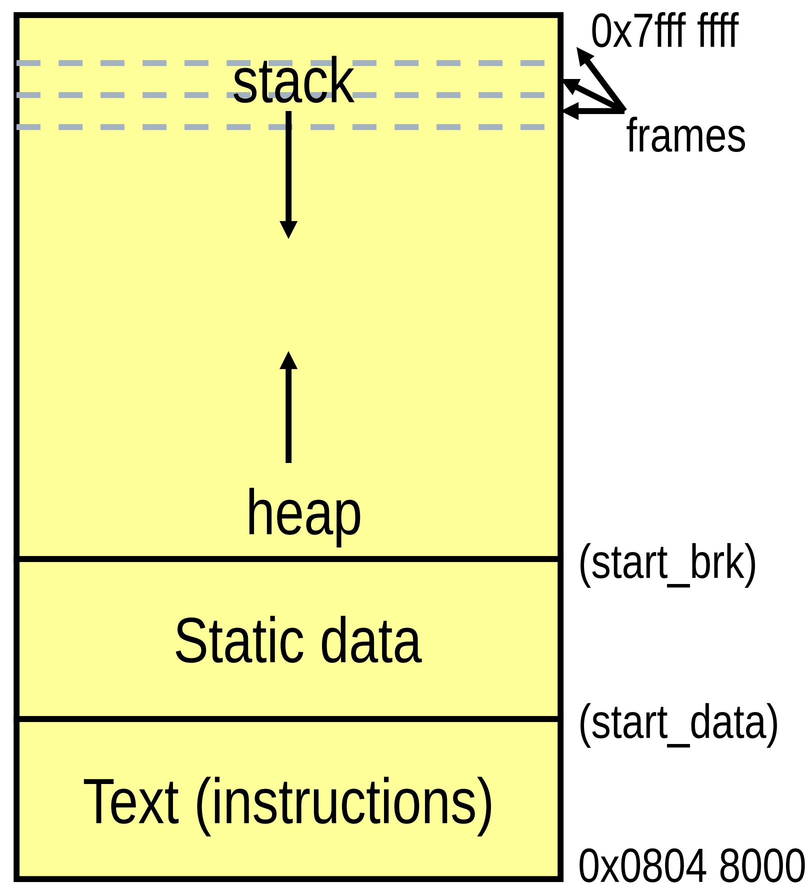
ARM (Linux) Memory Map

Stack: starts at 0x7fff ffff and grows down to lower addresses. Bottom of the stack resides in the SP register

Heap: starts above static (page aligned) and grows up to higher addresses. Allocation done explicitly with malloc(). Deallocation with free(). Runtime error if no free memory before running into SP address.

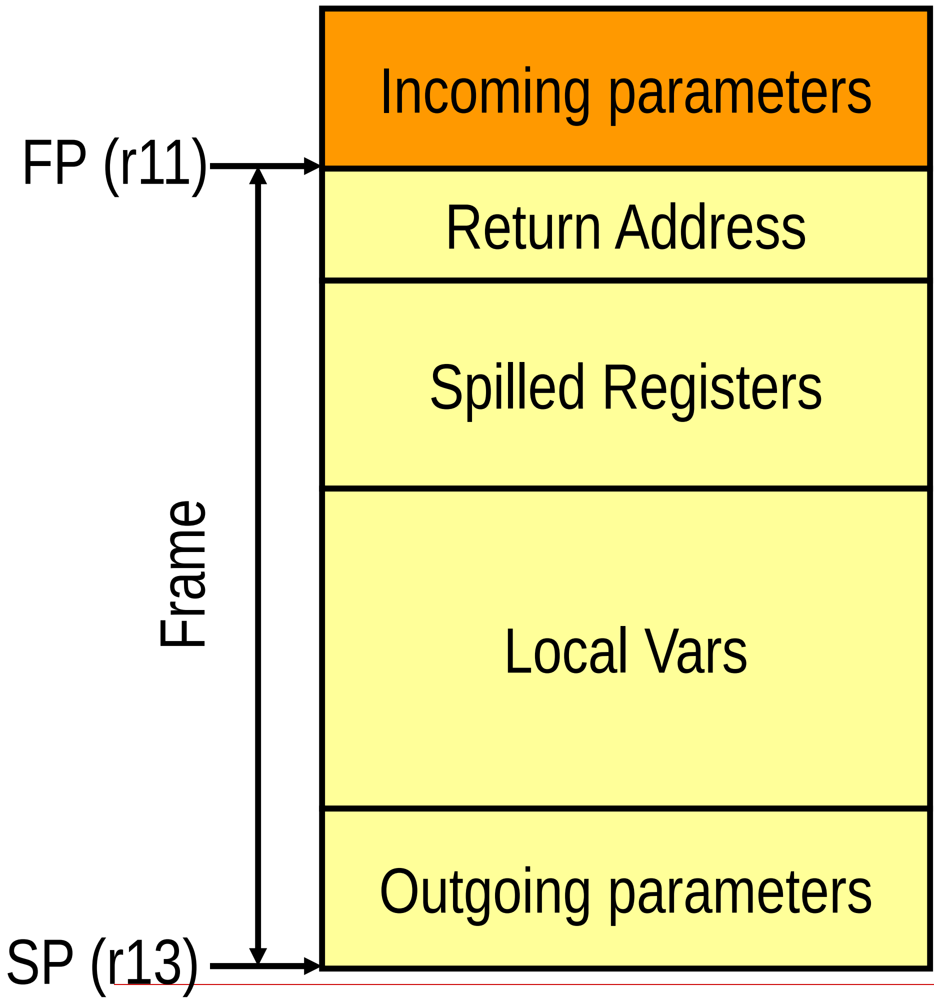
Static: starts above text (page aligned). Holds all global variables and those locals explicitly declared as "static".

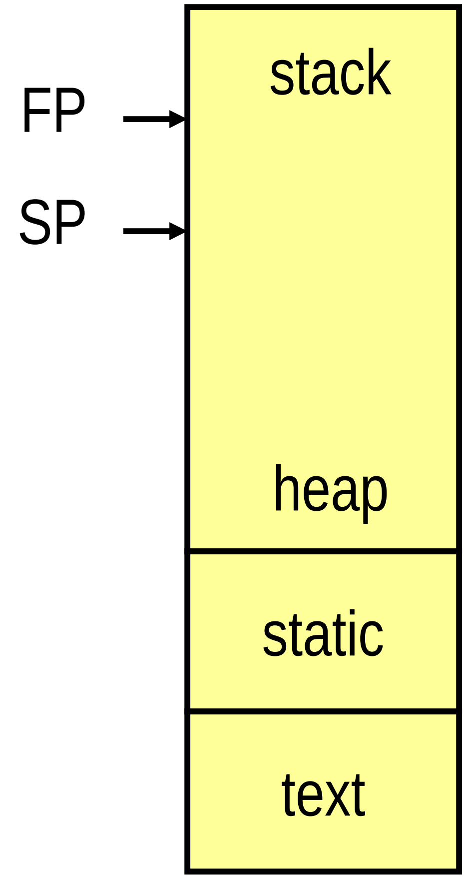
Text: starts at 0x08048000. Holds all instructions in the program (except for Dynamically linked library routines DLLs)



The ARM Stack Frame

FUNCTION CALLS





Allocating space to local variables

- Local variables (by default) are created when you enter a function, and disappear when you leave
  - Technical terminology: local variables are placed in the automatic storage class (as opposed to the static storage class used for globals).

- Automatics are allocated on the call stack
  - How?
    by incrementing (or decrementing) the pointer to the top of the call stack

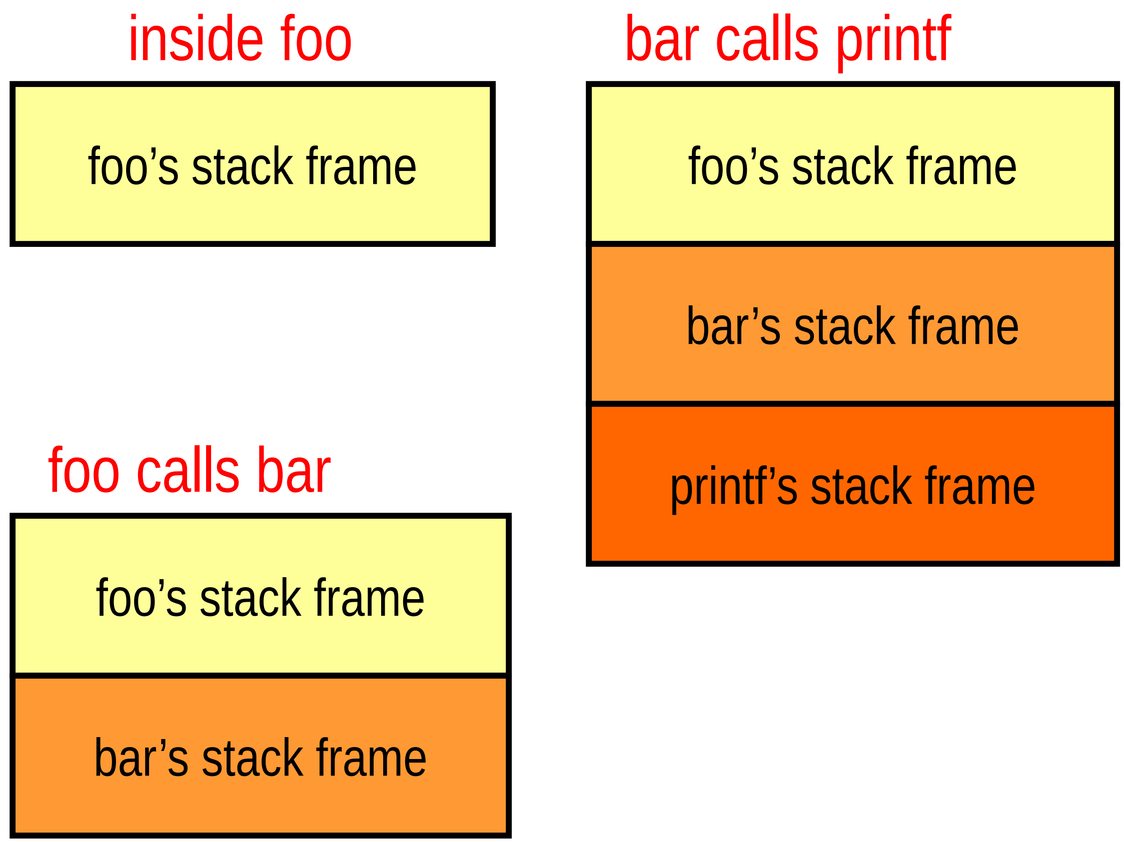
  - sub r13, r13, #12 // SP = SP – 12, allocate space for 3 integer locals
    add r13, r13, #12 // SP = SP + 12, de-allocate space for locals

FUNCTION CALLS

The stack grows as functions are called

```
void foo()
{
   int x, y[2];
   bar(x);
}

void bar(int x)
{
   int a[3];
   printf();
}
```

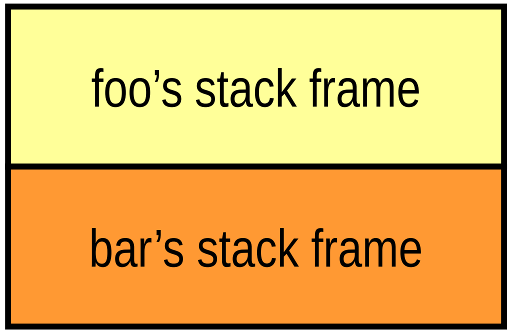


The stack shrinks as functions return

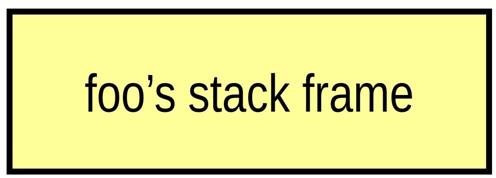
```
void foo()
{
   int x, y[2];
   bar(x);
}

void bar(int x)
{
   int a[3];
   printf();
}
```

printf returns



bar returns



Stack frame contents

FUNCTION CALLS

```
void foo()
{
   int x, y[2];
   bar(x);
}

void bar(int x)
{
   int a[3];
   printf();
}
```

inside foo – foo's stack frame

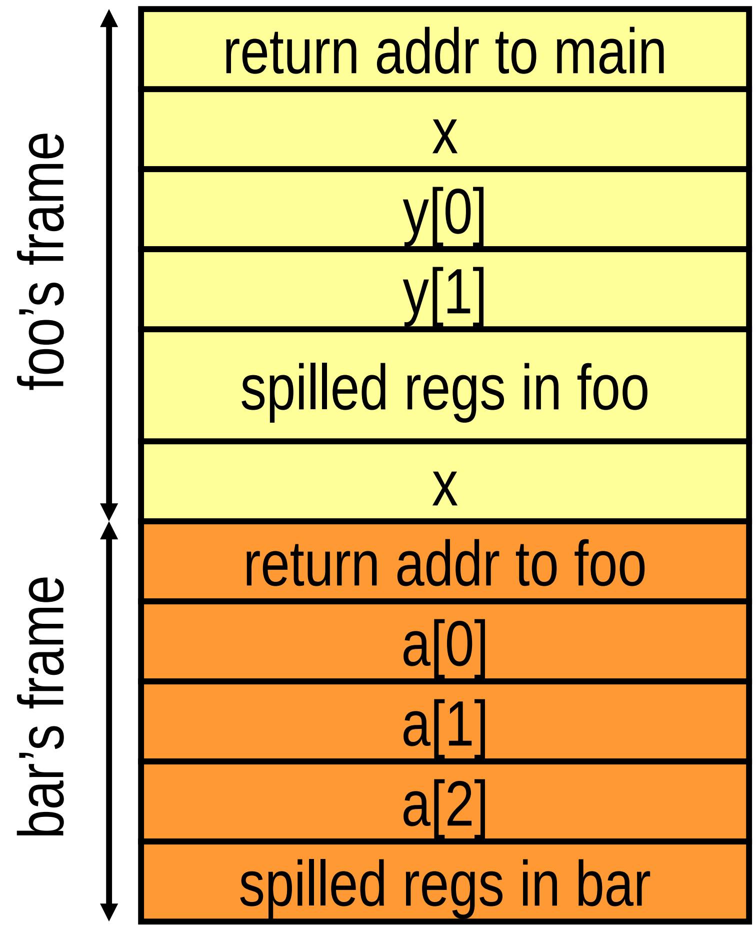
return addr to main
x
y[0]
y[1]
spilled regs in foo

Stack frame contents (2)

```
void foo()
{
    int x, y[2];
    bar(x);
}

void bar(int x)
{
    int a[3];
    printf();
}
```

foo calls bar



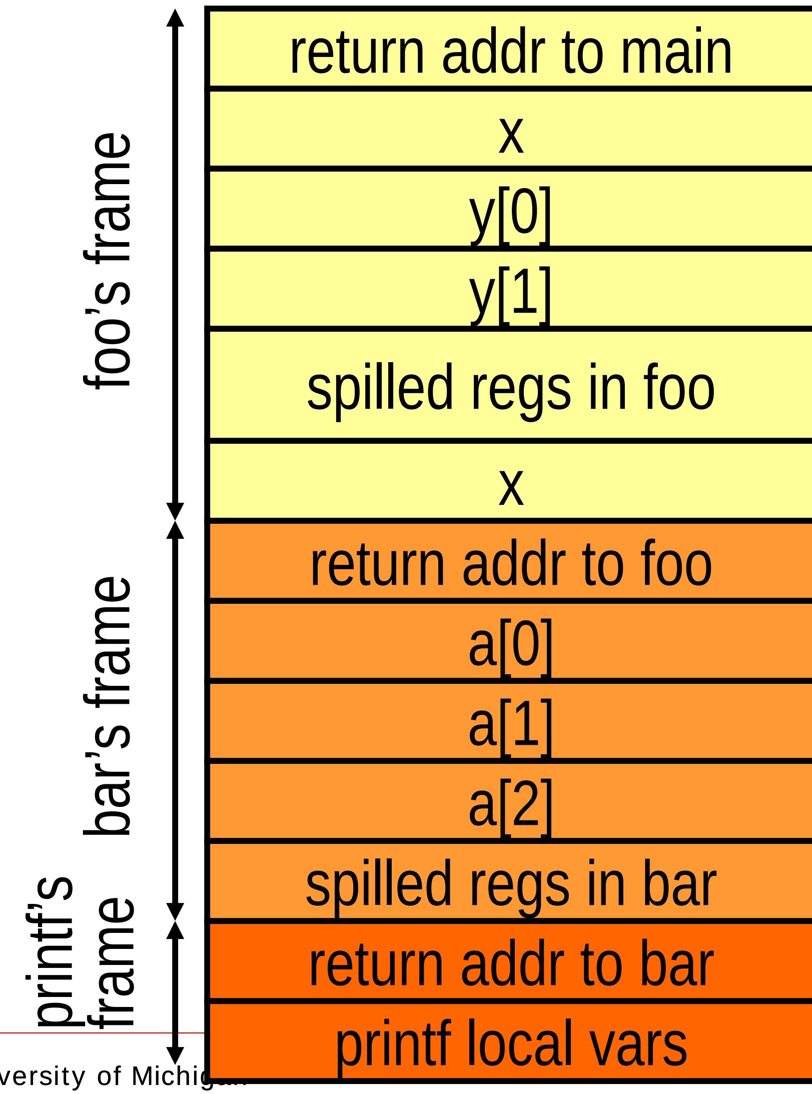
FUNCTION CALLS

Stack frame contents (3)

bar calls printf

```
void foo()
{
   int x, y[2];
   bar(x);
}

void bar(int x)
{
   int a[3];
   printf();
}
```



Recursive function example

FUNCTION CALLS

```
main()
{
    foo(2);
}

void foo(int a)
{
    int x, y[2];
    if (a > 0)
        foo(a-1);
}
```

	return addr to …
main calls foo	2
	return addr to main
	x, y[0], y[1]
	spills in foo
foo calls foo	1
	return addr to foo
	x, y[0], y[1]
	spills in foo
foo calls foo	0
	return addr to foo
	x, y[0], y[1]
	spills in foo

Virtual functions

FUNCTION CALLS

- Call stack is <u>identical</u>
- key difference: call is implemented as table lookup (i.e., indirect call versus direct call)
  - "class->method()" is translated to:

```
"switch (class.type)
        case A:  r1 = &class.method_typeA
        case B:  r1 = &class.method_typeB
        etc.
call r1"
```

Assigning variables to memory spaces

FUNCTION CALLS

`int w;`	w goes in static, as it's a global
`void foo(int x)`	x goes on the stack, as it's a parameter
`{`	
`static int y[4];`	y goes in static, 1 copy of this!!
`char *p;`	p goes on the stack
`p = malloc(10);`	allocate 10 bytes on heap, ptr set to the address
`…`	
`printf("%s\n", p);`	string goes in static, pointer to string on stack, p goes on stack
`}`