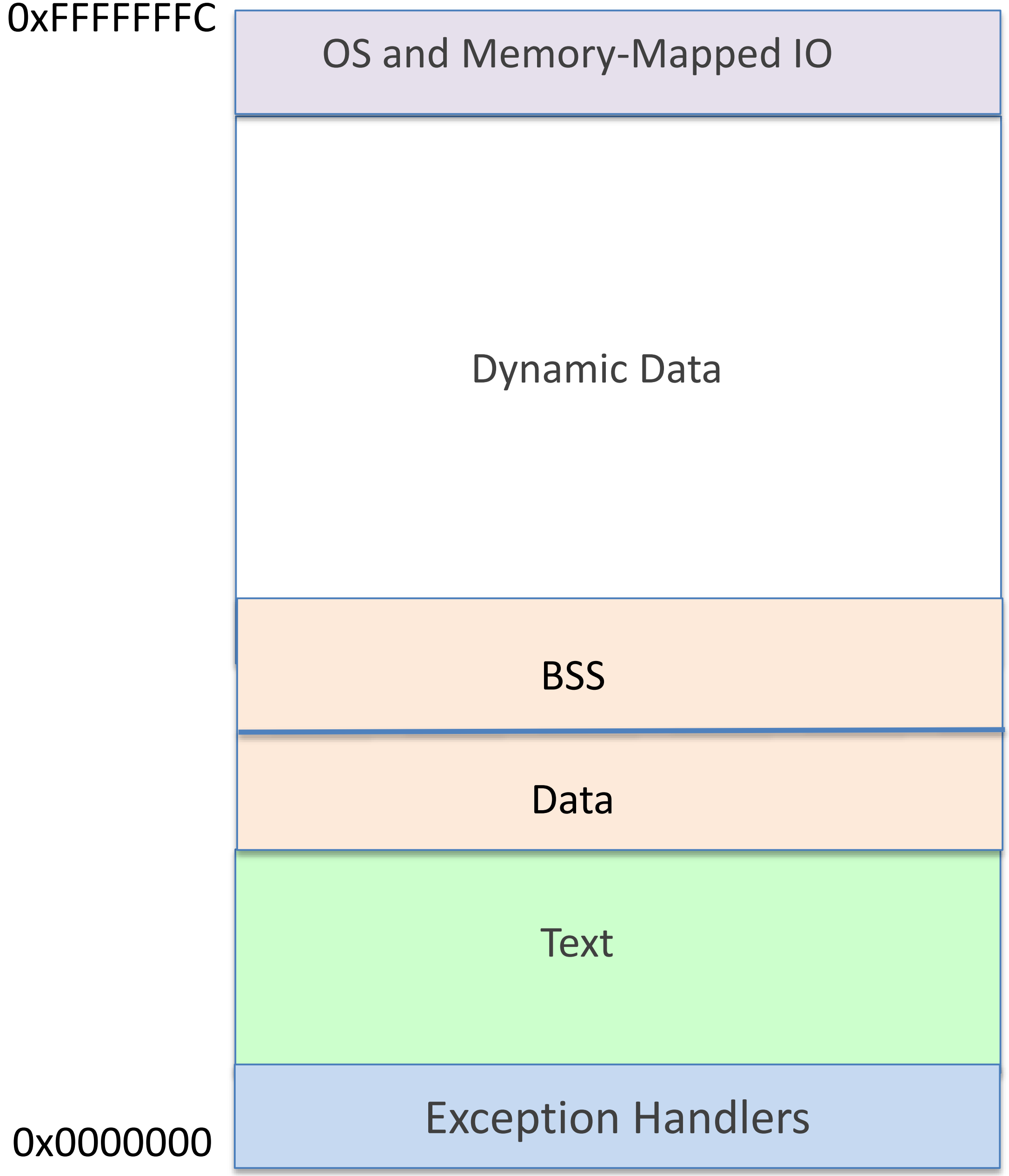
CSE 30: Computer Organization and Systems Programming

Lecture 11:
Memory map of C programs
ARM Programmer's Model

Diba Mirza
University of California, San Diego

Typical ARM Memory Map



Program Memory Map

- “Text” (instructions in machine language)
- “Data” contains any global or static variables which have a pre-defined value and can be modified. That is any variables that are not defined within a function (and thus can be accessed from anywhere) or are defined in a function but are defined as *static* so they retain their value across subsequent calls.
- “BSS” also known as *uninitialized data*, is usually adjacent to the data segment. The BSS segment contains all global variables and static variables that are initialized to zero or do not have explicit initialization in source code
- “Heap” (for dynamically allocated data)
- “Stack” (for function local variables)

Heap and stack change in size as the program executes

Viewing the memory map with 'size'

- You can use the size command to check the memory map of your executable

mem.c:

```
void foo() {
}
```

```
 text    data     bss     dec     hex filename
   20       0       0      20      14 hello.o
```

Checking the memory map

```
void foo() {
    int i=10;  //line 1
}
```

Output of size before adding line 1

text	data	bss	dec	hex	filename
20	0	0	20	14	hello.o

The size of which section of the memory map of the program will increase on adding line 1?

A. Text

B. Data

C. BSS

D. None of the above

E. All of the above

Checking the memory map

```
void foo() {
    static int i=10;  //line 1
}
```

Output of size before adding line 1

```
text    data    bss    dec    hex filename
  20       0      0     20     14 hello.o
```

The size of which section of the memory map of the program will increase on adding line 1?

A. Text
B. Data
C. BSS
D. None of the above
E. All of the above

Checking the memory map

```
int i=0;  //line 1
void foo() {
}
```

Output of size before adding line 1

```
text    data    bss    dec    hex filename
  20       0      0     20     14 hello.o
```

The size of which section of the memory map of the program will increase on adding line 1?

A. Text
B. Data
C. BSS
D. None of the above
E. All of the above

Translations

- High-level language program (in C)

```
swap (int v[], int k)
{      int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

one-to-many

C compiler

- Assembly language program (for MIPS)

```
swap:  sll $2, $5, 2
       add $2, $4, $2
       lw  $15, 0($2)
       lw  $16, 4($2)
       sw  $16, 0($2)
       sw  $15, 4($2)
       jr  $31
```

one-to-one

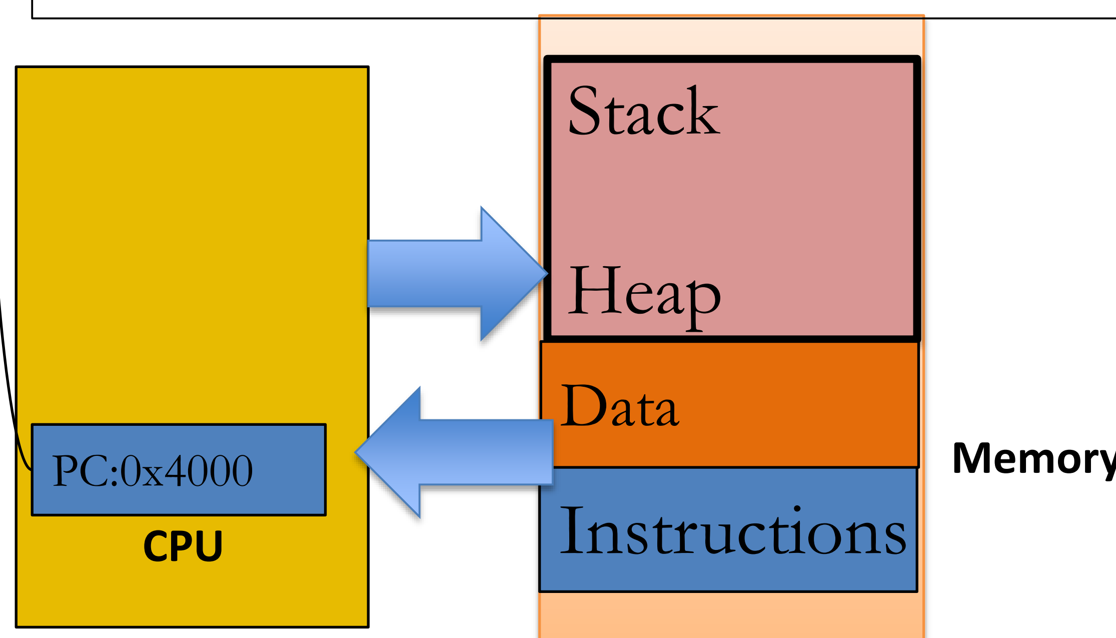
assembler

- Machine (object, binary) code (for MIPS)

```
000000 00000 00101 0001000010000000
000000 00100 00010 0001000000100000
 . . .
```

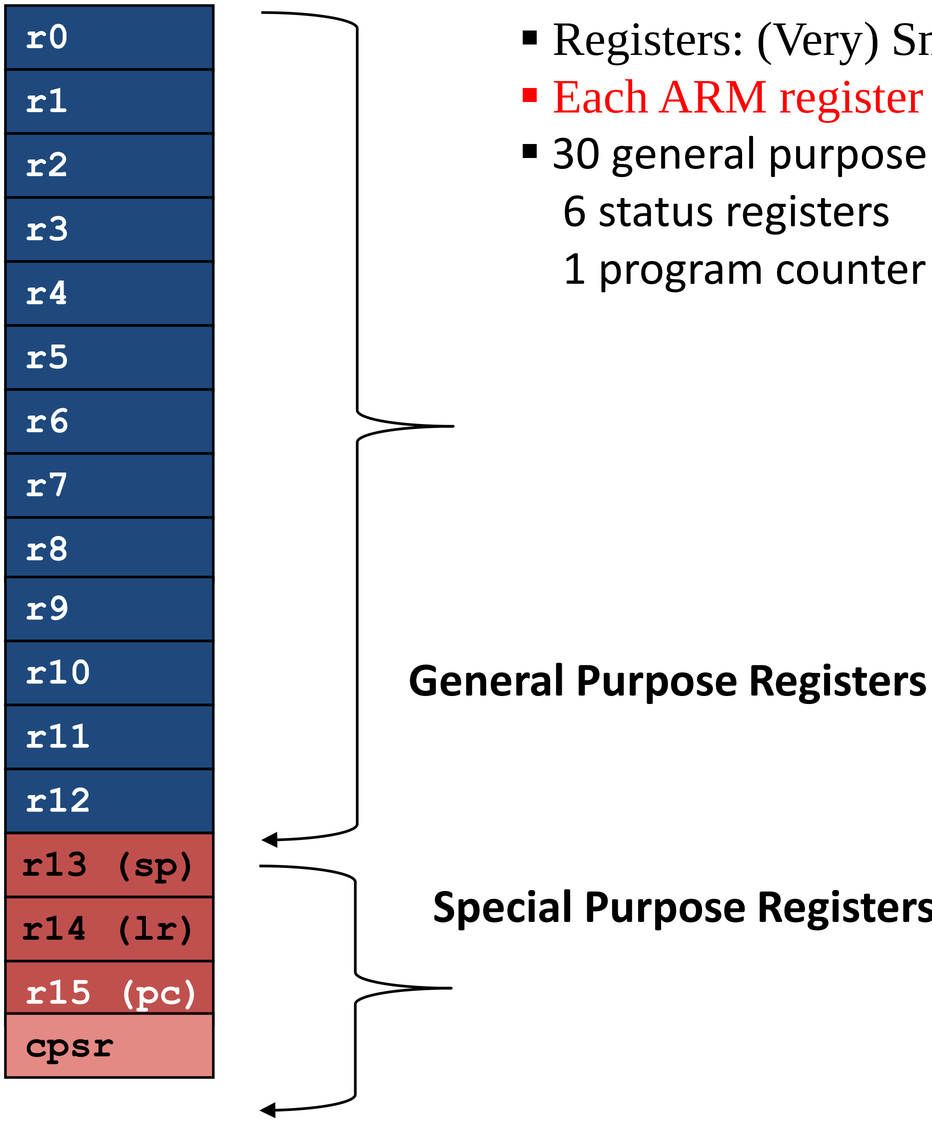
Steps in program execution

```
0x4000 Swap:  mov     r2, r5
0x4004        add     r4, r2, #1
0x4008        ldr     r10, [r6,r2]
0x400c        ldr     r11, [r6,r4]
0x4010        mov     r1, r10
0x4014        str     r11,[r6,r2]
0x4018        str     r1, [r6,r4]
0x401c        bx      lr
```



The ARM Register Set

- Registers: (Very) Small amount of memory inside the CPU
- Each ARM register is 32 bits wide
- 30 general purpose registers
  6 status registers
  1 program counter



ARM Assembly Variables: Registers

- Unlike HLL like C or Java, assembly cannot use variables
  - Why not? Keep Hardware Simple
- Data is put into a register before it is used for arithmetic, tested, etc.
- Result is stored in a register (later stored to memory)
- Benefit: Since registers are directly in hardware, they are very fast
- In C (and most High Level Languages) variables declared first and given a type
  - Example:
    ```
    int fahr, celsius;
    char a, b, c, d, e;
    ```
- Each variable can ONLY represent a value of the type it was declared as (cannot mix and match `int` and `char` variables)
- In Assembly Language, the registers have no type; operation determines how register contents are treated

PA2 ARM Skeleton code

```
.syntax unified
.text
.align 8
.global get_min_ARM
.func get_min_ARM, get_min_ARM
.type get_min_ARM, %function

get_min_ARM:
    @ Save caller's registers on the stack
    push {r4-r11, ip, lr}

    @ YOUR CODE GOES HERE (list *ls is in r0)
    @-----------------------
    @ (your code)
    @ put your return value in r0 here:
    @-----------------------
    @ restore caller's registers
    pop {r4-r11, ip, lr}
    @ ARM equivalent of return
    BX lr
.endfunc
.end
```

Basic Types of Instructions

1. Arithmetic: Only involves processor and registers
   - compute the sum (or difference) of two registers, store the result in a register
   - move the contents of one register to another

2. Memory Instructions: Transfer of data between registers and memory
   - load a word from memory into a register
   - store the contents of a register into a memory word

3. Control Transfer Instructions: Change flow of execution
   - jump to another instruction
   - conditional jump (e.g., branch if register == 0)
   - jump to a subroutine

Arithmetic Instructions

In C:

a=b+c;

In ARM:

ADD r0, r1, r2

Specifying constants in Arithmetic Instructions

In C:

```
a=b+10;
```

In ARM:

```
ADD r0, r1, #10
```

- Immediates are numerical constants.
- They appear often in code, so there are ways to indicate their existence

How big can immediates be?

Q: What is a plausible range for the immediate in the instruction ADD r0, r1, <immediate>?

A. 0 to $(2^{32}-1)$

B. 0 to 255

Assignment Instructions

In C:

a=b;

a=10;

In ARM:

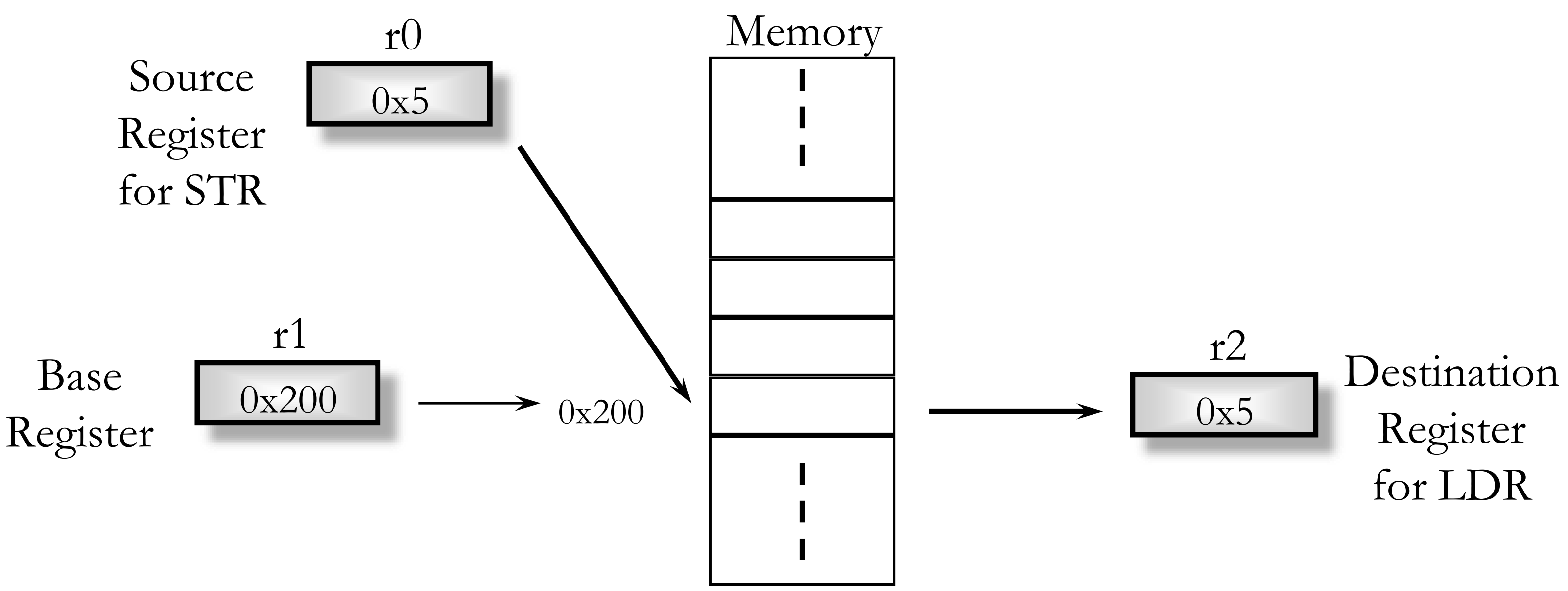
Data transfer (memory) Instructions

- Separate instructions to transfer data between registers and memory:
  - Memory to register (load)
  - Register to memory (store)

- Load/store usage (Base register addressing mode)

Base Register Addressing Mode

- The memory location to be accessed is held in a base register
  - `STR r0, [r1] @Store contents of r0 to location pointed to`
    `@ by contents of r1.`
  - `LDR r2, [r1] @Load r2 with contents of memory location`
    `@pointed to by contents of r1.`



Data Transfer Instructions

In C:

```
void foo (int *p){
    *p=10;
}
```

In ARM:

Data Transfer Instructions

In C:

```
void foo (int *p){
   int a=*p;
}
```

In ARM: