Lecture 9: Data Transfer Instructions

CSE 30: Computer Organization and Systems Programming

Diba Mirza
Dept. of Computer Science and Engineering
University of California, San Diego

UCSD

Addressing modes

i. Base register addressing

ii. Base displacement

a) Pre indexed:

a) Pre indexed:

a) Pre indexed:

Data Transfer: Memory to Register

```
LDR r2,[r1, #12]
```

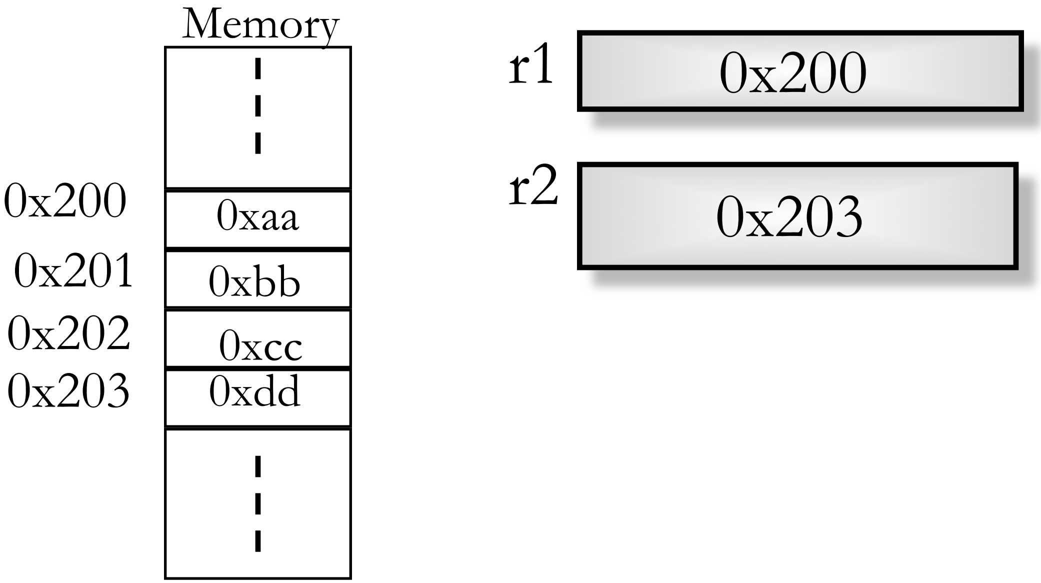
Given the value of r2 and r1 below, the above instruction stores four bytes starting at which memory location into r2

A. 0x200

B. 0x212

C. 0x20C

D. None of the above



Data Transfer: Memory to Register

```
STR r2,[r1, #-4]!
```

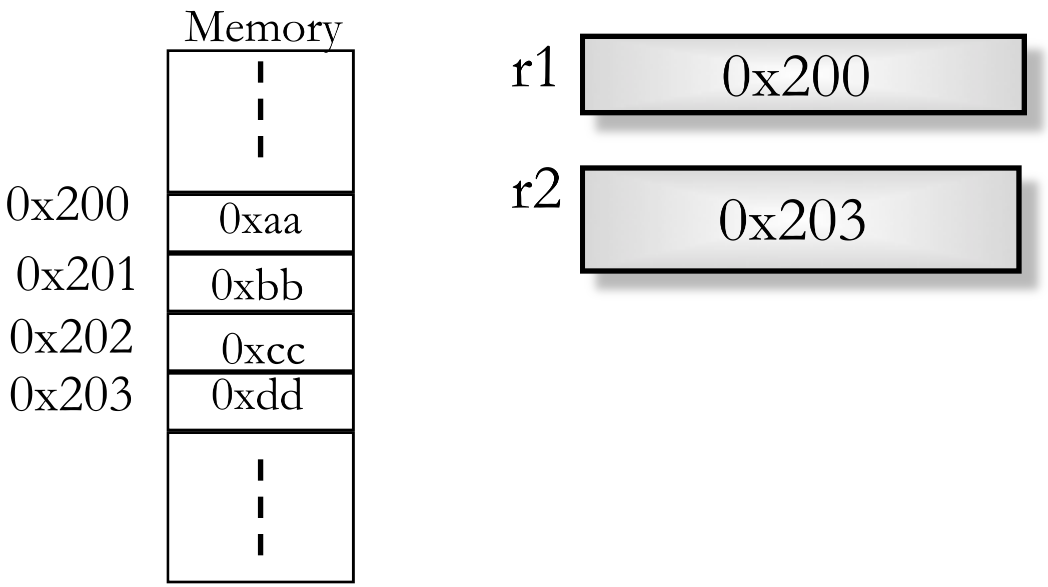
What are the contents of r1 after the above instruction is executed?

A. 0x200

B. 0x1fc

C. 0x204

D. r1 is unchanged



Memory

0x200 0xaa

0x201 0xbb

0x202 0xcc

0x203 0xdd

r1 0x200

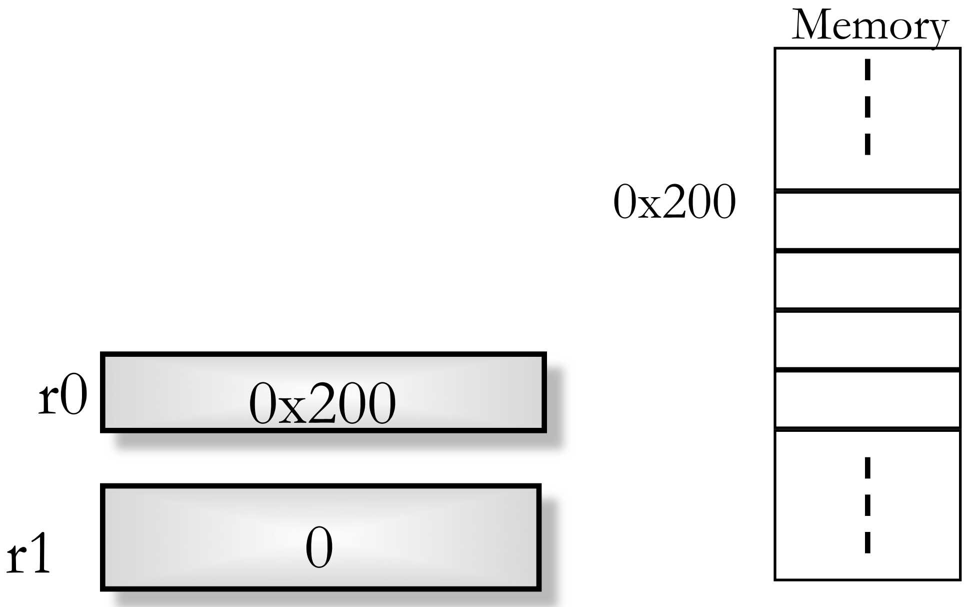
r2 0x203

b) Post-indexed:

b) Post-indexed:

b) Post-indexed:

Accessing arrays with LDR/STR



Compile by hand

- g=h+A[8]

Jump!

Topic : ARM Procedures

CSE 30: Computer Organization and Systems Programming

Diba Mirza
Dept. of Computer Science and Engineering
University of California, San Diego

UCSD

C functions

CalleR: the calling function
CalleE: the function being called

```
main() {
 int a,b,c;
 ...
 c = sum(a,b);/* a,b,c:r0,r1,r2*/
 ...
 }

/* sum function */
int sum(int x, int y) {
   return x+y;
 }
```

C functions

Steps in function call

```
main() {
 int a,b,c;
 ...
 c = sum(a,b);

 ...
 }



/* sum function */
int sum(int x, int y) {
  return x+y;
 }
```

Steps needed for function call & return

1. Transfer control to the function being called (callee) (This is some location in memory different from the current address in pc)
2. Pass parameters to the function
3. Transfer control back to the caller once function execution is complete
4. Make return values available to caller function

Let's focus on the transfer of control to and from the function called

Instruction Support for Functions

C

```
 ... sum(a,b);... /* a,b:$s0,$s1 */
 }
int sum(int x, int y) {
  return x+y;
 }
```

A
R
M

address
1000
1004
1008
1012
1016

In ARM, all instructions are stored in memory just like data. So here we show the addresses of where the programs are stored.

UCSD

Using the branch instruction….

C

```
... sum(a,b);... /* a,b:r4,r5 */
}
int sum(int x, int y) {
  return x+y;
}
```

A
R
M

```
address
 1000  ...
 1004  ...
 1008  ...
 1012 B sum           ; branch to sum
 1016 return_loc:...
 1020 ...
 2000 sum: ADD r0,r0,r1
 2004 B return_loc
```

Is there something wrong with using the simple branch instruction ?

A. Yes
B. No

Using the branch instruction….

C

```
... sum(a,b);... /* a,b:r4,r5 */
}
int sum(int x, int y) {
  return x+y;
}
```

ARM

```
address
 1000  ...
 1004  ...
 1008  ...
 1012 B sum ;branch to sum
 1016 return_loc:...
 1020 ...
 2000 sum: ADD r0,r0,r1
 2004 B return_loc
```

Is there something wrong with using the simple branch instruction ?

A. Yes
B. No

Reason: `sum` might be called by many functions, so we can't return to a fixed place.
The calling proc to `sum` must be able to say"return back here"somehow.

Instruction Support for Functions

C

```
... sum(a,b);... /* a,b:r4,r5 */
}
int sum(int x, int y) {
  return x+y;
}
```

ARM

address

```
1000 ...
1004 ...
1008 MOV  lr,1016      ; lr = 1016
1012 B    sum        ; branch to sum
1016 ...
1020 ...
2000 sum: ADD r0,r0,r1
2004 BX lr  ; MOV  pc,lr i.e., return
```

Instruction Support for Functions

Single instruction to jump and save return address: jump and link (BL)

- Before:

```
1008 MOV lr, 1016  ; lr=1016
1012 B sum         ; go to sum
```

- After:

```
1008 BL sum   # lr=1012, goto sum
```

Why have a BL? Make the common case fast: function calls are very common. Also, you don't have to know where the code is loaded into memory with BL.

Instruction Support for Functions

- Syntax for `BL` (branch and link) is same as for `B` (branch):

```
    BL <label>
```

- `BL` functionality:
  - –Step 1 (link): Save address of *next* instruction into `lr` (Why next instruction? Why not current one?)
  - –Step 2 (branch): Branch to the given label

Instruction Support for Functions

- Syntax for `BX` (branch and exchange):

```
BX register
```

- Instead of providing a label to jump to, the `BX` instruction provides a register which contains an address to jump to
- Only useful if we know exact address to jump
- Very useful for function calls:
  - `BL` stores return address in register (`lr`)
  - `BX lr` jumps back to that address

How to pass arguments to a function?

Passing arguments & return values

ARM

```
main() {

 int a=10,b=20,c;
 c = sum(a,b);




 }



/* sum function */
int sum(int x, int y) {
   return x+y;
 }
```

Passing arguments & return values

```
main() {

 int a=10,b=20,c;
 c = sum(a,b);



 }
```

If the value of 'a' is stored in r0 before the function call, does this value remain the same after the call to sum returns?

A. Yes

B. No

```
/* sum function */
int sum(int x, int y) {
   return x+y;
 }
```

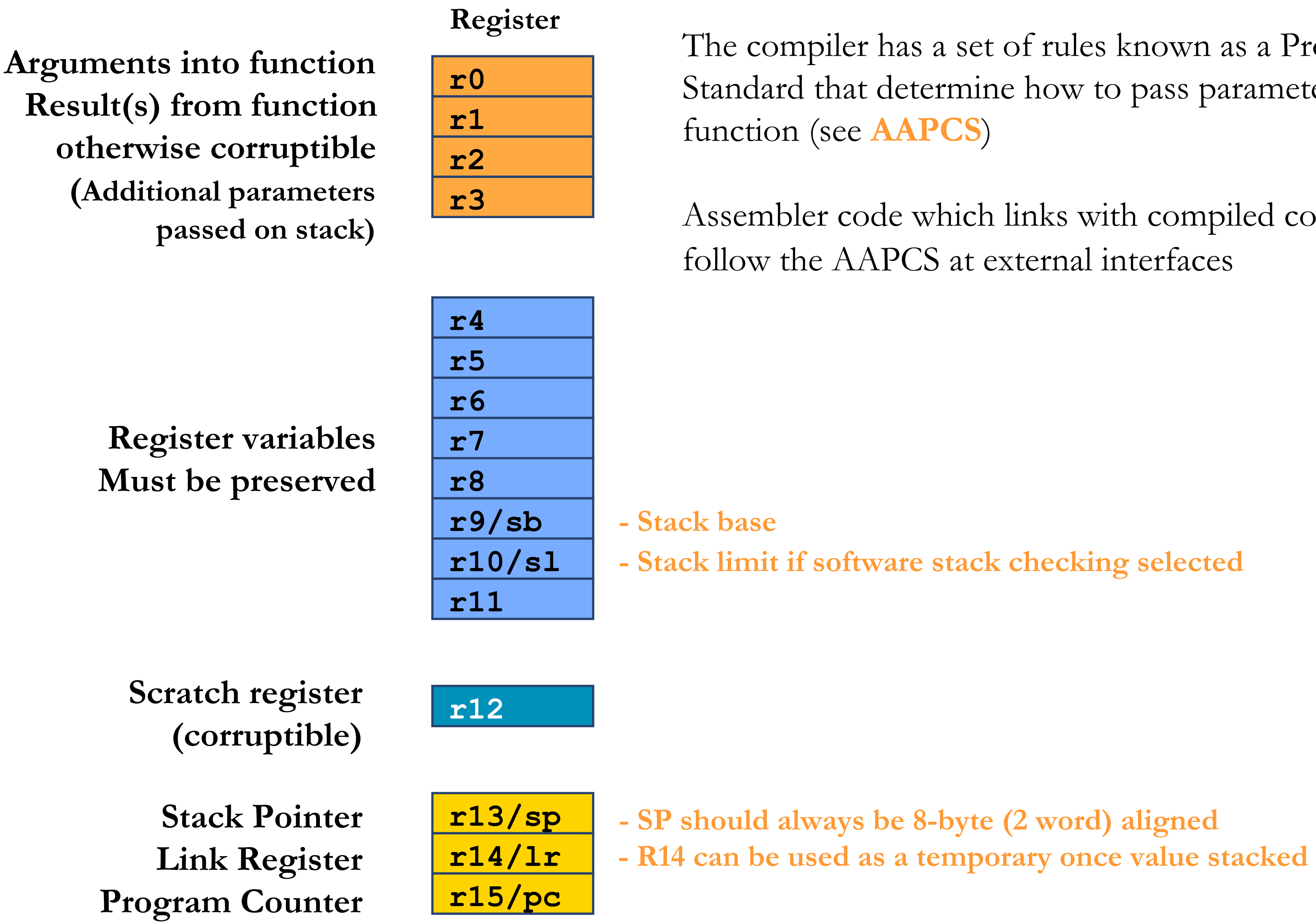
Register Conventions

- Register Conventions: A set of generally accepted rules as to which registers are guaranteed to be unchanged after a procedure call (`BL`) and which may be changed.

Arm Procedure Call Std.

The compiler has a set of rules known as a Procedure Call Standard that determine how to pass parameters to a function (see **AAPCS**)

Assembler code which links with compiled code must follow the AAPCS at external interfaces



Register

Arguments into function
Result(s) from function
otherwise corruptible
(Additional parameters passed on stack)

`r0`
`r1`
`r2`
`r3`

Register variables
Must be preserved

`r4`
`r5`
`r6`
`r7`
`r8`
`r9/sb` - Stack base
`r10/sl` - Stack limit if software stack checking selected
`r11`

Scratch register (corruptible)

`r12`

Stack Pointer
Link Register
Program Counter

`r13/sp` - SP should always be 8-byte (2 word) aligned
`r14/lr` - R14 can be used as a temporary once value stacked
`r15/pc`

Passing arguments & return values

ARM

```
main() {

 int a,b,c;
 .........

  c = sum(a,b);



  }
```

```
/* sum function */
int sum(int x, int y) {
    return x+y;
  }
```