CSCI 6461 Computer Systems Architecture

Addressing Memory

ELF Memory Map

The segments of a program in execution

Executable and Linkable Format (ELF)

- ELF (Executable and Linkable Format) is a file format for executables, libraries, and object code, defining memory layout and linking.
- An ELF (Executable and Linkable Format) number of sections varies depending on the file's purpose (e.g., executable, shared library, or object file) and how it was compiled.
- When a program is loaded into memory it gains additional "dynamic" segments

ARM32 ELF and unix-like systems

- ARM32 binaries are typically compiled into ELF files,
  - this define how the code and data are organized for a executable or linkable file
- ELF is a flexible format supports various processor-specific details,
  - such as ARM's instruction set and memory alignment requirements,
  - machine type field (e.g., EM_ARM for ARM architecture).

C program example

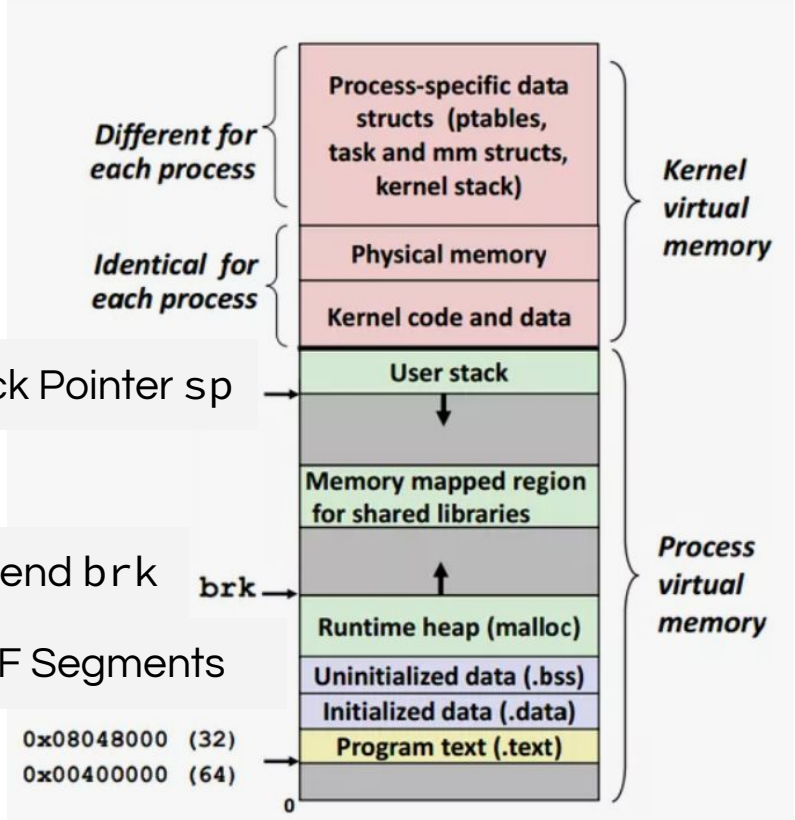
From the course repo - a simple C program to show the various ELF segments using `objdump -c -S a.out`

```c
/*
 * simple.C
 * a file to show the layout of objects
 * in the a.out executable
 * using objdump -s -C
 * */
#include <stdio.h>

// initialized global variable (r/w)
int global_var1 = 100;

// uninitialized global variable (r/w)
int global_var2;

// initialized global variable (r/w) string
char global_var3[] = "This is a global string";

int main(int argc, char* argv[]) {
        int local_var = 99;
        // the reference to prinf should be mentioned in the .dynstr
        // the string itself ("Globals are ...") is r/o in .data
        printf("Globals are %d %d %s\n", global_var1, global_var2, global_var3);
        return 0;
}
```

https://github.com/jzburns/csci-comp-arch/blob/master/C/simple.C

The Linux Virtual Memory Map

When the ELF executable becomes a **process** , it gains a number of additional memory resources, such as stack, heap and share library region



ARM32 Memory Map Sections

- **Kernel Space** : 1 GB (1024 MB) reserved for the kernel.
- **Stack**: Thread stacks, grows downward
- **Memory Mapping** : Shared libraries, mapped files.
- **Heap**: Dynamic memory, grows upward
- **BSS**: Uninitialized global/static variables (zeroed).
- **Data**: Initialized global/static variables.
- **Text** (Code): Program's executable code (read-only).

Text and Data Segments

- The **text** segment stores the machine language program.
  - In addition to code, also literals (constants) and read-only data
- The global **data** segment stores global variables that, in contrast to local variables, can be accessed by all functions in a program (Read/Write segment)

Dynamic Data Segment

- The **dynamic data** segment holds the **stack** and the **heap**.
  - The data in this segment is not known at start-up but is dynamically allocated and deallocated throughout the execution of the program
- On start-up, the operating system sets up the stack pointer (sp) to point to the top of the stack.
  - The stack typically grows downward.
- The stack includes temporary storage and local variables, such as arrays, that do not fit in the registers

Addressing Memory

Modes for accessing memory

What are ‘Addressing Modes’?

- Data is rarely hard-coded into an instruction using immediates (eg `mov r0, #2`).
- We nearly always read our data from physical memory (RAM),
  - then we can use the data, and
  - write it back to physical memory
- There are differnt methods (or modes) we can use to achieve the correct outcome

Memory Addressing Modes

- Register indirect
  - `ldr r1, [ r0 ]`
- Register indirect with offset
  - `ldr r1, [ r0, #4 ]`
- Double Register Indirect
  - `ldr r1, [ r0, r2 ]`

in these examples, `r0` is the **base** register

Initializing the base pointer in C

In C, we are all familiar with the following approach:

So `ptr` is our base pointer (or base address)

```c
#include <stdio.h>

int array[6] = { 2,3,11,4,55,6 };

int main(int argc, char* argv[]) {
  int *ptr = array;
  *(ptr += 4) += 10;
  printf("New number: %d\n", *ptr);
  return 0;
}
```

https://github.com/jzburns/csci-comp-arch/blob/master/C/simple.C

Initializing the base register in ARM

```
//Step 1 - declare a .data section (like in the ELF)
//Step 2 - add your integer array
//Step 3 - pick a base register (eg, r0) and use the =
//operator to load the address

.data
  array: .word 2,3,11,4,55,6
ldr r0, =array
```

```
int array[6] = { 2,3,11,4,55,6 };
int *ptr = array;
```

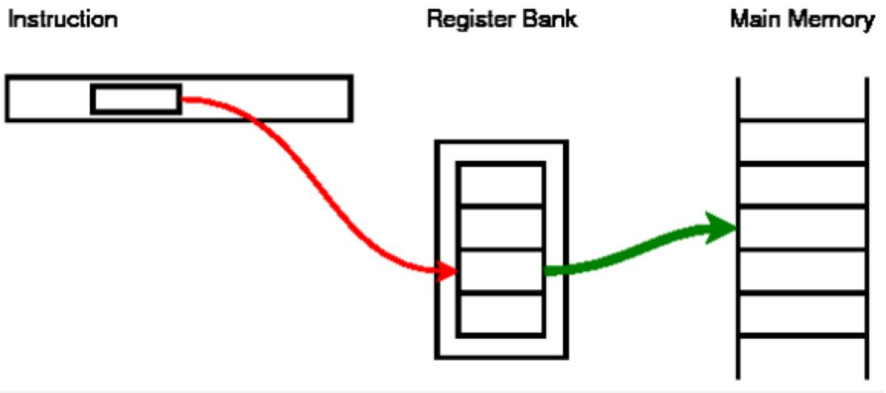
Register Indirect Addressing

ARM has register indirect addressing e.g. loading a register from a memory location:

```
ldr r2, [ r0 ]
```

We can think of `r0` as being the address of the **first** element in the array pointed to by `r0`

Register Indirect Addressing



- It takes only a few bits to select a register (4 bits in the case of ARM . . . `r0-r15`)
- A register can (typically) hold an arbitrary address (32 bits in the case of ARM)

Indirect with Offset

ARM allows offsets of **12 bits** in `ldr/str`

This may be added or subtracted, for example:

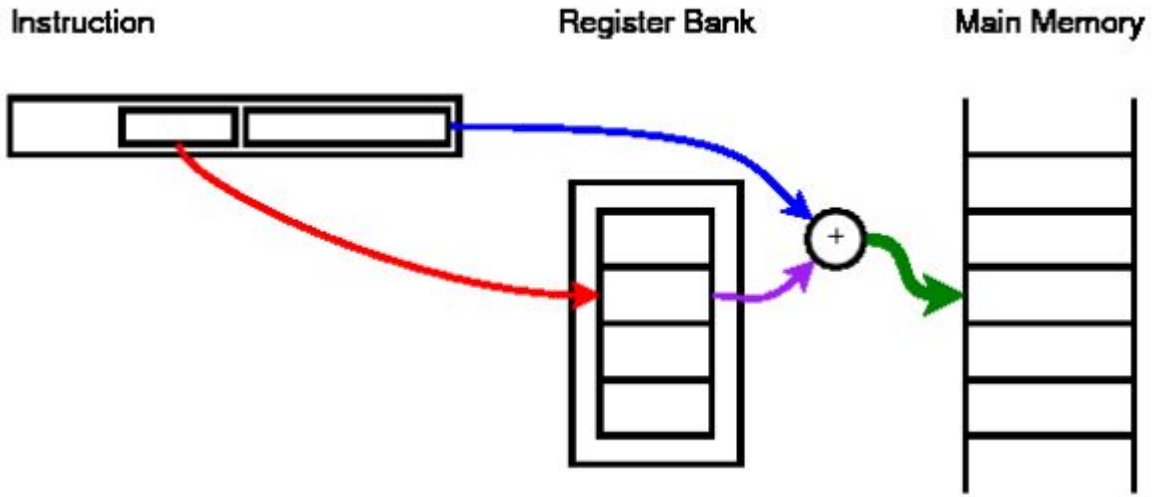
```
ldr r0, [ r1, #8 ]
str r3, [ r6, #-64 ]
```

This provides a range of ± ~4 Kbytes around a 'base' register - adequate for most purposes.

Indirect with Offset

The address is calculated from a register value and a literal

The register specifier is just a few bits
The offset can be ‘fairly small’
With one register ‘pointer’ any of several variables in nearby addresses may be addressed



Address Arithmetic

We can operate on registers, so we can:

- store/load/move addresses
- do arithmetic to calculate addresses

Rather than using extra `add` instructions, we often use

Base + Offset Addressing

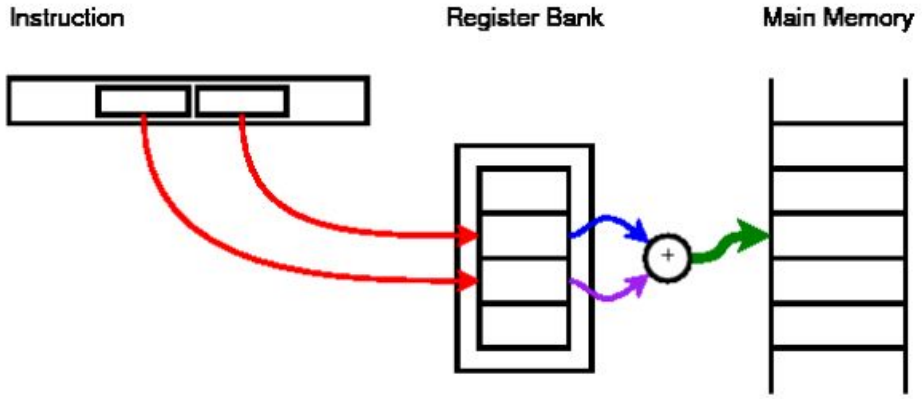
- address addition done within the operand

Double Register Indirect

ARM also allows offsets using a second register

This may be added or subtracted, for example:

```
ldr r0, [ r1, r2 ]
str r3, [ r6, -r4 ]
```



PC + Offset Addressing


```
start:
    ldr r0, =b
    ldr r1, [ r0 ]
    add r2, r1, #5
    str r2, [ r0 ]

.data
    b: .word 20
    c: .word 30
```

The PC is the address of the current instruction plus 8 bytes (2 words). This means

`ldr r0, =b` is actually: `ldr r0, [ pc, #8 ]`

What effect does the code have?

Why does the ARM PC register point to the instruction after the next one to be executed? - Stack Overflow

Pointer arithmetic

```
*(ptr += 4) += 10;
```

How can we implement this in ARM32 assuming `r0` is our base register? Let's try:

```
add r0, r0, #16
ldr r5, [ r0 ]
add r5, r5, #10
str r5, [ r0 ]
```

Iteration over an array

Now we can use `lsl`

Adding 10 to each element in an array

```
.global _start
_start:
      ldr r0, =array1       // array base pointer
      mov r1, #0            // offset
      mov r2, #0            // loop counter

loop:
      cmp r2, #6
      bge end
      ldr r3, [r0, r1]      // read base + offset to reg
      add r3, r3, #10       // add 10 to the value
      str r3, [r0, r1]      // write reg to base + offset
      add r2, r2, #1        // inc loop
      lsl r1, r2, #2        // calc offset
      b loop
end:
      svc #2

.data

array1: .word 2,3,11,4,55,6
```

A note on Incrementing & offsets

Effect of LSL R2, R1, #2

Itr	R1	R2	Dec
0	0000	0000	0
1	0001	0100	4
2	0010	1000	8
3	0011	1100	12
4	0100	10000	16

Operations within an instruction

Iteration optimization from the previous example:

```
ldr r3, [ r0, r2, lsl #2 ]
add r2, r2, #1
```

Thus we can eliminate the `lsl r1, r2, #2`

The highlighted code does **not** change the value of `r2`

Register Byte Instructions

`ldrb / strb`

Strings

- Java:
  - `String message= "Hello";`
- ARM:

```
.data
    message: .asciz "Hello World!"
```

Iterating over an array of "chars"

Suppose we want to iterate ocer each character individually:

```
for (int i= 0; i<message.length(); i++) {
   // do something with this:
   message.charAt(i);
}
```

Accessing characters: LDRB, STRB


```
ldr r1, =message
ldrb r0, [r1]     // fetch 1st byte
ldrb r0, [r1,#1]  // fetch 2nd byte
ldrb r0, [r1,#2] // fetch 3rd byte
```

get next character from the string


```
for (int i= 0; . . . ; i++) {
   System.out.print(message.charAt(i));
}
```

Use a second register (e.g. R2) to hold `i` around loop:

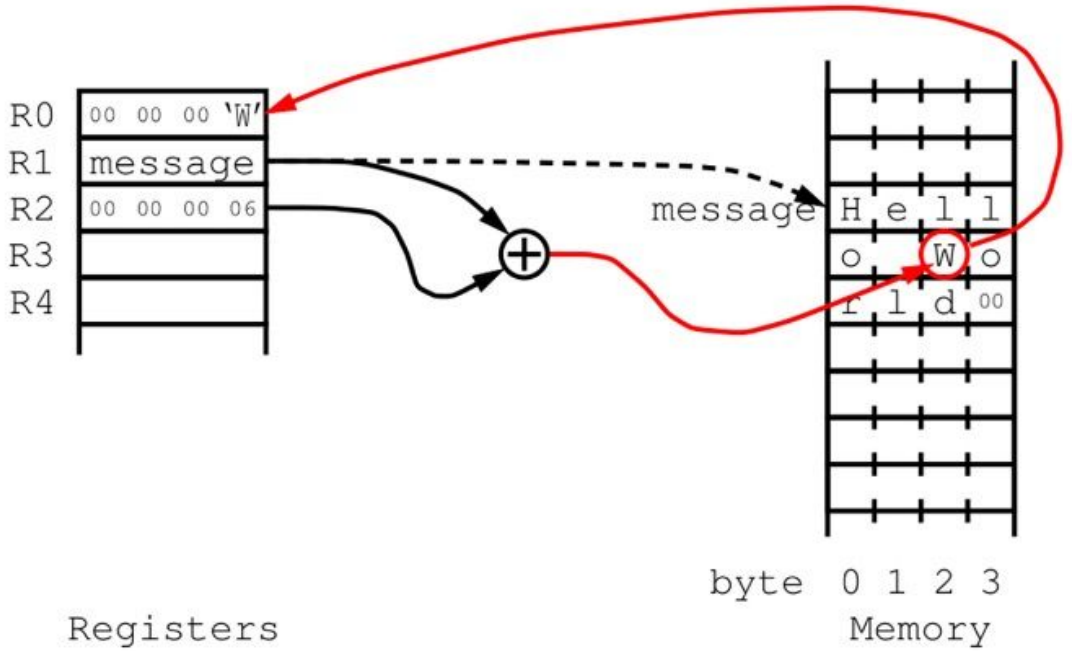
First attempt, no loop


```
mov r2, #0          //int i= 0
ldr r0, =message    //beginning address of ‘message’
...
ldrb r0, [r1,r2]    // message.charat(i) = r1+r2
add r2, r2, #1      //  i++
...
```

ldrb Load Register Byte (immediate) loads a byte from memory, zero-extends it, and writes the result to a register.

get next character from the string

Part way through execution



get next character from the string

optimisation: avoid using both r1 and r2 for addresses

actually change the address in r1

```
ldr r0, =message
...
ldrb r1, [ r0 ]
add r0, r0, #1
```

Indexing Options

3 main types: offset, pre and post

Indexing modes

In addition to scaling the index register, ARM provides

- offset addressing
- pre-indexed addressing
- post-indexed addressing

Mode	ARM Assembly	Address	Base Register
Offset	`LDR R0, [R1, R2]`	R1 + R2	Unchanged
Pre-index	`LDR R0, [R1, R2]!`	R1 + R2	R1 = R1 + R2
Post-index	`LDR R0, [R1], R2`	R1	R1 = R1 + R2

get next character from the string

- Optimisation: avoid using `add r1, r1, #1`
- Change the address in `r1` using "**post-indexed** " operand form

```
ldr r1, =message
ldrb r0, [r1],#1
```

while not at end of string

Every Java/C String knows how long it is. But how do we know in assembly when the end of the string is reached?

```
.data
    message: .asciz "Hello World!"
```

Then iterate over the array of characters until 0 is reached. In C/C++ this is sometimes referred to as the **null** character.

Whole loop


```
ldr r0, =message
loop:
    ldrb r1, [ r0 ],#1
    cmp r1, #0
    beq end
    b loop
end: // end here
.data
    message: .asciz "Hello World!"
```

Further Examples

(simple) indirect:

```
ldr r0, [ r1 ] ;
```
r0 ← value at [r1], r1 unchanged

(base +) offset:

```
ldr r0, [ r1, #4 ] //
```
r0 ← value at [r1+4], r1 unchanged

```
ldr r0, [ r1, r2 ] //
```
r0 ← value at [r1+r2], r1 unchanged

post-indexed:

```
ldr r0, [ r1 ], #4 //
```
r0 ← value at [r1], r1 ← r1+4

```
ldr r0, [ r1 ], r2 //
```
r0 ← value at [r1], r1 ← r1+r2

pre-indexed:

```
ldr r0, [ r1, #4 ]! //
```
r0 ← value at [r1+4], r1 ← r1+4

```
ldr r0, [ r1, r2 ]! //
```
r0 ← value at [r1+r2], r1 ← r1+r2