Addressing

Steven R. Bagley

Loads and Stores

- So far we've seen `LDR` and `STR` in use to access a specific address, e.g.
  `LDR R0, one`
- On other systems, this is knows as *absolute addressing*
- But ARM doesn't support absolute addressing…
- Again the assembler is helping us out here…

Indirect Addressing

- ARM only supports *Indirect Addressing*
- Works just like a pointer in C
- A register contains the address to look the value up in
- CPU gets the value in memory at the address in the register

Indirect Addressing

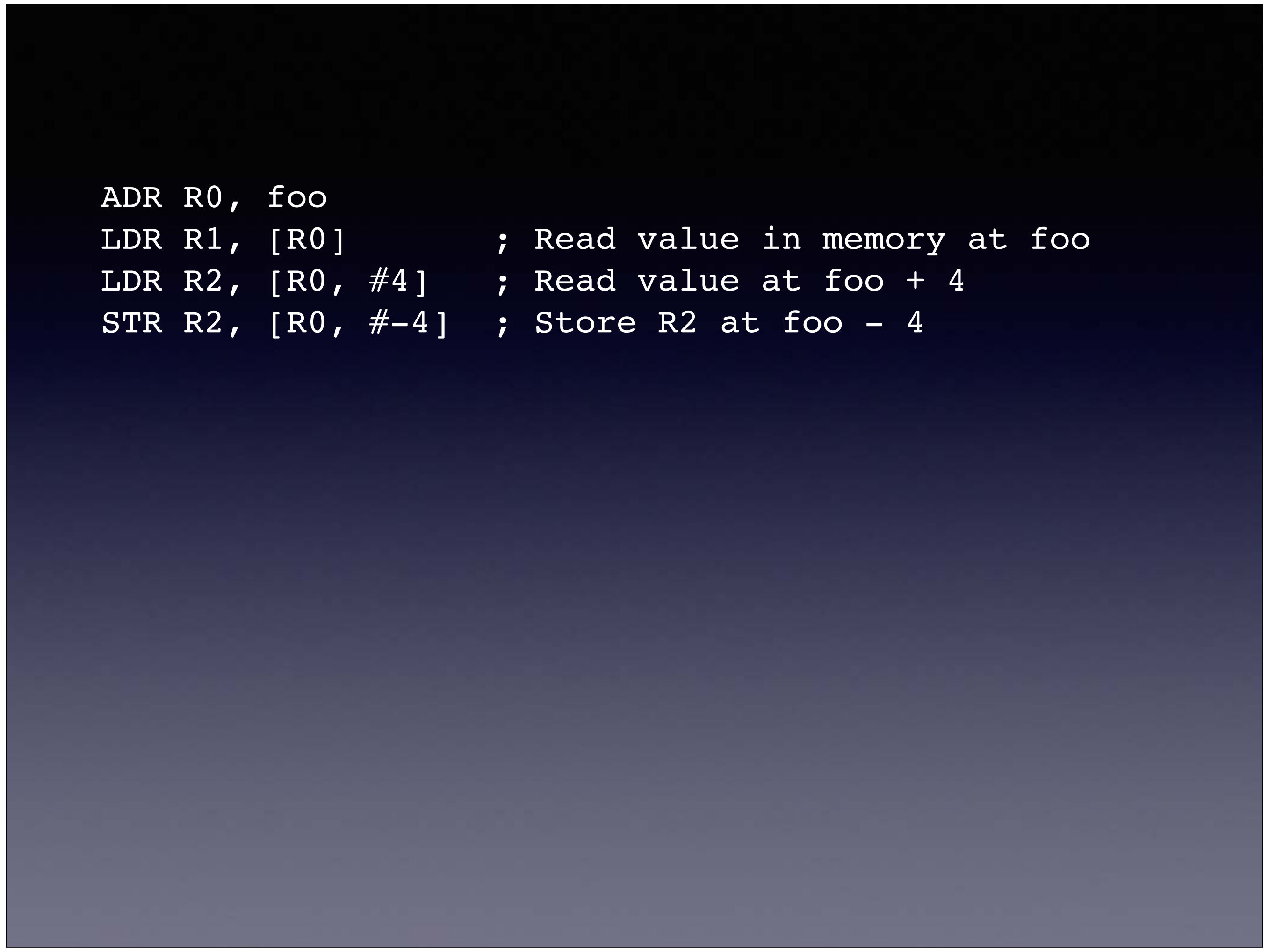
- Load the register with the address
  ```
  ADR R1, mesg
  ```
- Then use that register in square brackets to access the value
  ```
  LDRB R0, [R1]
  STR R4, [R2]
  ```
- If we change the value in the register we change the address looked up

Absolute Addressing

- So how does the assembler handle ‘absolute addressing’
- Does it generate the address in a register and do an indirect lookup on that?
- Not quite…
- Uses the `PC` again…
- With an index offset

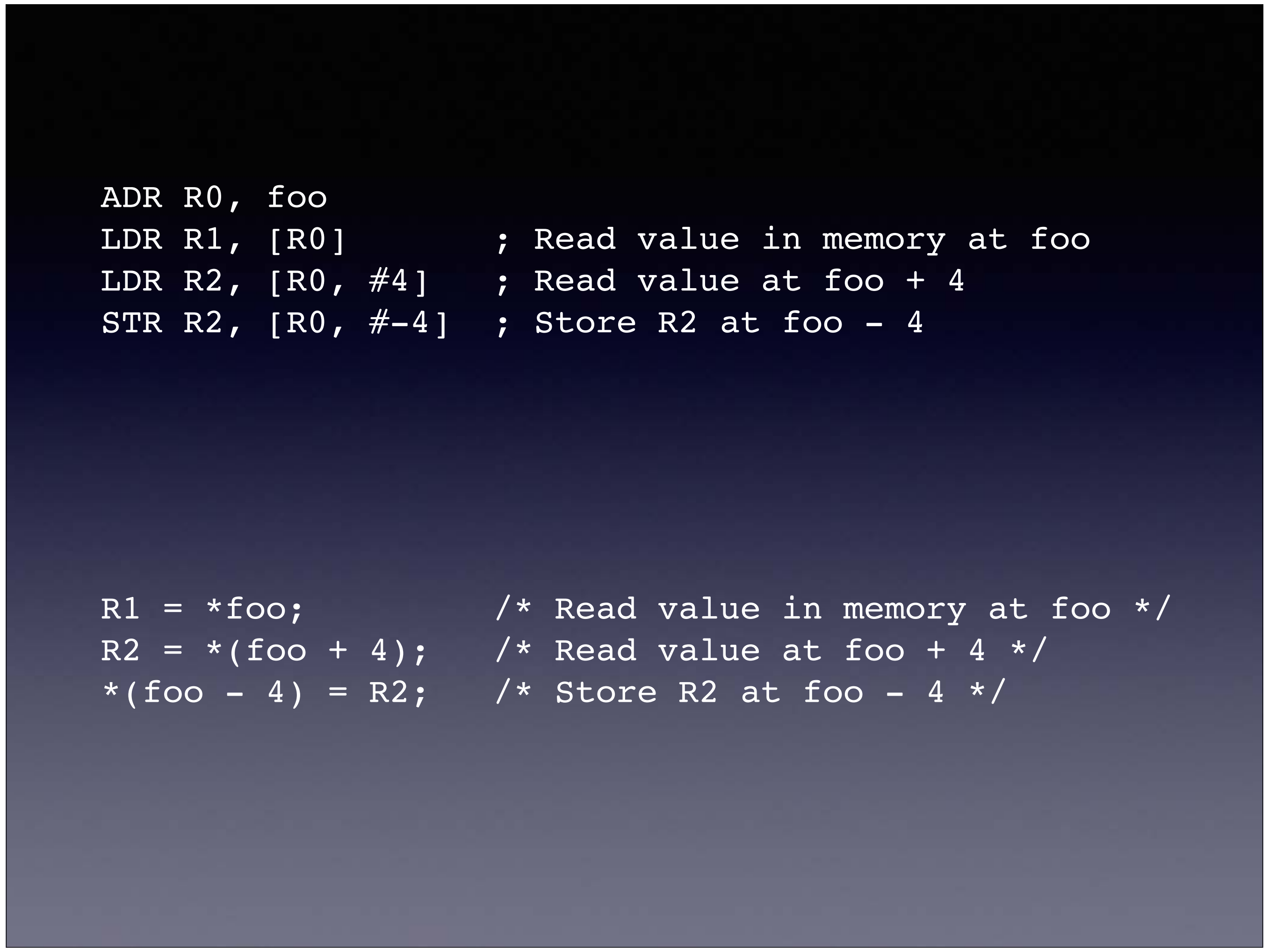
Indirect With Index

- The instruction layout for `LDR` and `STR` includes space for a 12-bit unsigned offset
- A further bit controls whether this is added or subtracted from the address in the register
- Yet another bit controls whether this happens before (pre-index) or after (post-index) the address is used to access memory



```
ADR R0, foo
LDR R1, [R0]       ; Read value in memory at foo
LDR R2, [R0, #4]   ; Read value at foo + 4
STR R2, [R0, #-4]  ; Store R2 at foo - 4
```

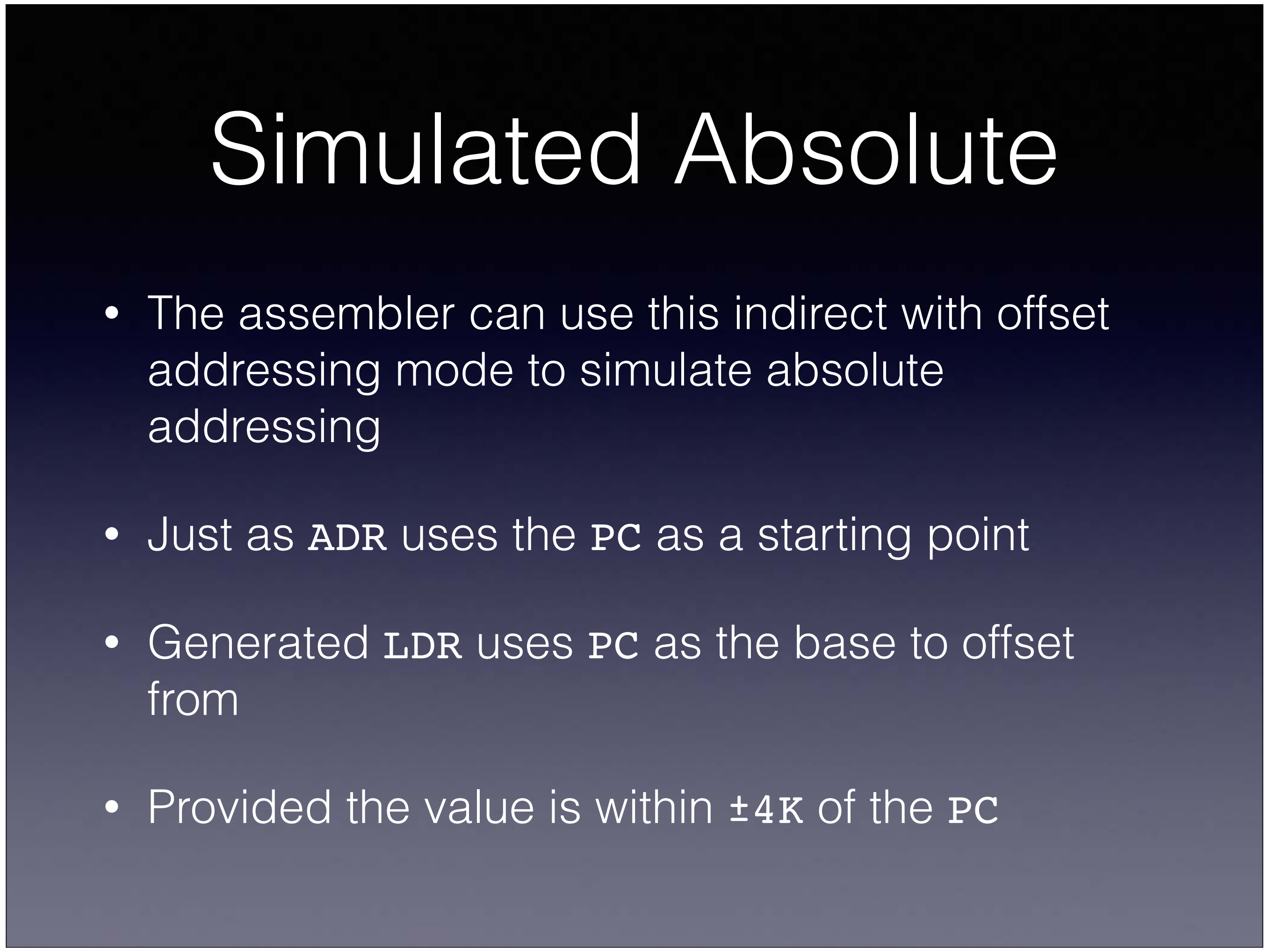
Good for accessing arrays…



```
ADR R0, foo
LDR R1, [R0]        ; Read value in memory at foo
LDR R2, [R0, #4]    ; Read value at foo + 4
STR R2, [R0, #-4]   ; Store R2 at foo - 4
```

```
R1 = *foo;          /* Read value in memory at foo */
R2 = *(foo + 4);    /* Read value at foo + 4 */
*(foo - 4) = R2;    /* Store R2 at foo - 4 */
```

Good for accessing arrays…



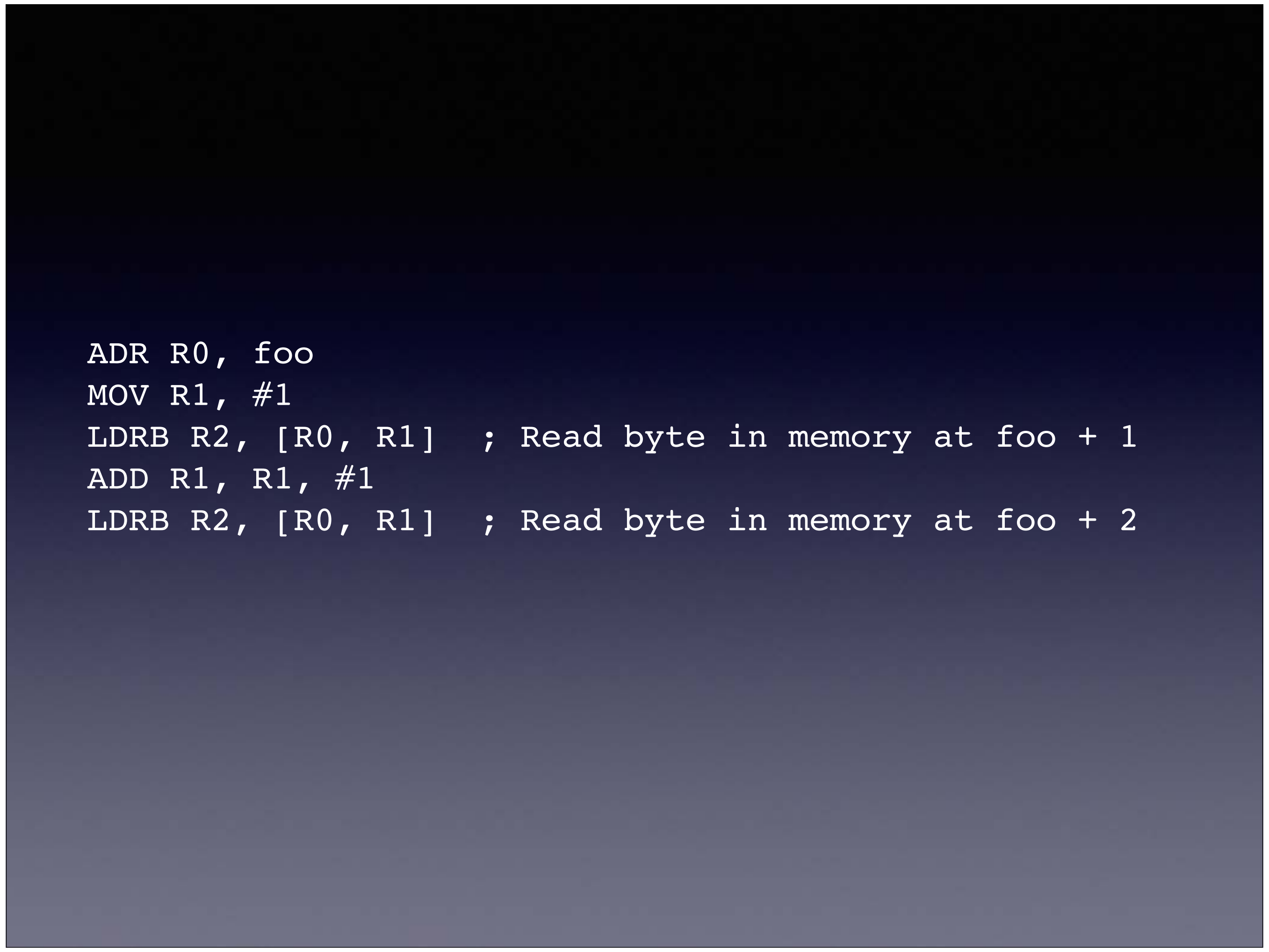
Simulated Absolute

- The assembler can use this indirect with offset addressing mode to simulate absolute addressing
- Just as **`ADR`** uses the **`PC`** as a starting point
- Generated **`LDR`** uses **`PC`** as the base to offset from
- Provided the value is within **`±4K`** of the **`PC`**

Otherwise instruction cannot be generated

Indirect with Register Index

- The index used does not have to be an immediate value
- Can also be another register…
- This gives us array like access to memory
- One register holds the base of the array, the other holds the index of the value we want to access



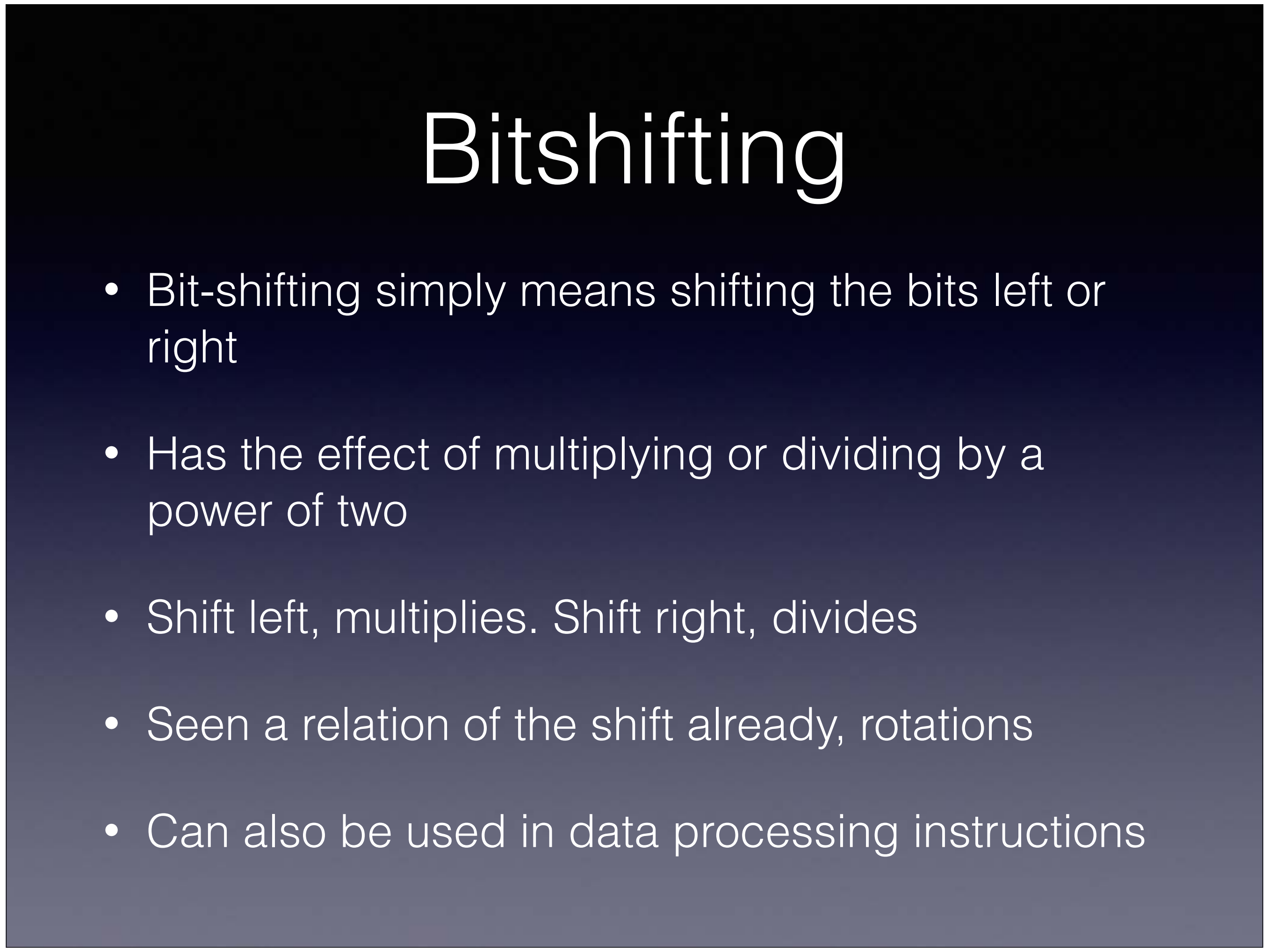
```
ADR R0, foo
MOV R1, #1
LDRB R2, [R0, R1]  ; Read byte in memory at foo + 1
ADD R1, R1, #1
LDRB R2, [R0, R1]  ; Read byte in memory at foo + 2
```

Again, good for accessing arrays…

Show how we can rewrite the strlen routine using indirect with register index addressing

Shifted register

- Register value added to address
- Treated as a *byte* offset
- Not scaled up to the size of the things (unlike C)
- However, we can specify that the register's value is shifted before it used.



Bitshifting

- Bit-shifting simply means shifting the bits left or right
- Has the effect of multiplying or dividing by a power of two
- Shift left, multiplies. Shift right, divides
- Seen a relation of the shift already, rotations
- Can also be used in data processing instructions

Just like shifting real numbers has the effect of multiplying or dividing by ten

Rotation:	0

rotates 0 places

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
																								H	G	F	E	D	C	B	A

Rotation:	1

rotates 2 places

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
B	A																									H	G	F	E	D	C

Rotation:	2

rotates 4 places

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
D	C	B	A																									H	G	F	E

Rotation:	12

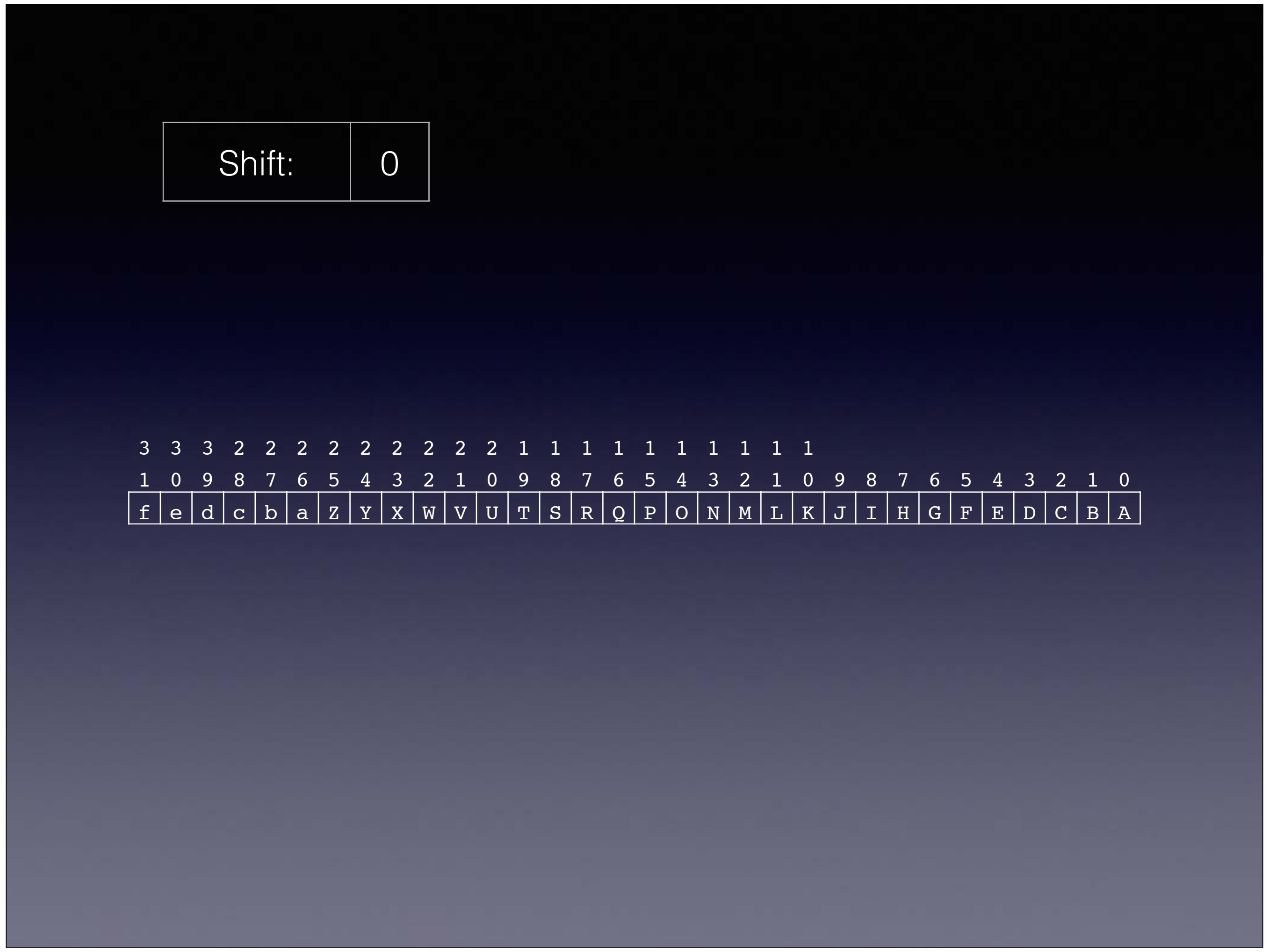
rotates 24 places

3	3	3	2	2	2	2	2	2	2	2	2	1	1	1	1	1	1	1	1	1	1										
1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0
																H	G	F	E	D	C	B	A								

Rotation:	15

rotates 30 places

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
																						H	G	F	E	D	C	B	A		



Shift:	0

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
f	e	d	c	b	a	Z	Y	X	W	V	U	T	S	R	Q	P	O	N	M	L	K	J	I	H	G	F	E	D	C	B	A

Shifting left

Shift:	1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
e	d	c	b	a	Z	Y	X	W	V	U	T	S	R	Q	P	O	N	M	L	K	J	I	H	G	F	E	D	C	B	A	0

Shift:	2

3	3	3	2	2	2	2	2	2	2	2	2	1	1	1	1	1	1	1	1	1	1										
1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0
d	c	b	a	Z	Y	X	W	V	U	T	S	R	Q	P	O	N	M	L	K	J	I	H	G	F	E	D	C	B	A	0	0

Shift:	12

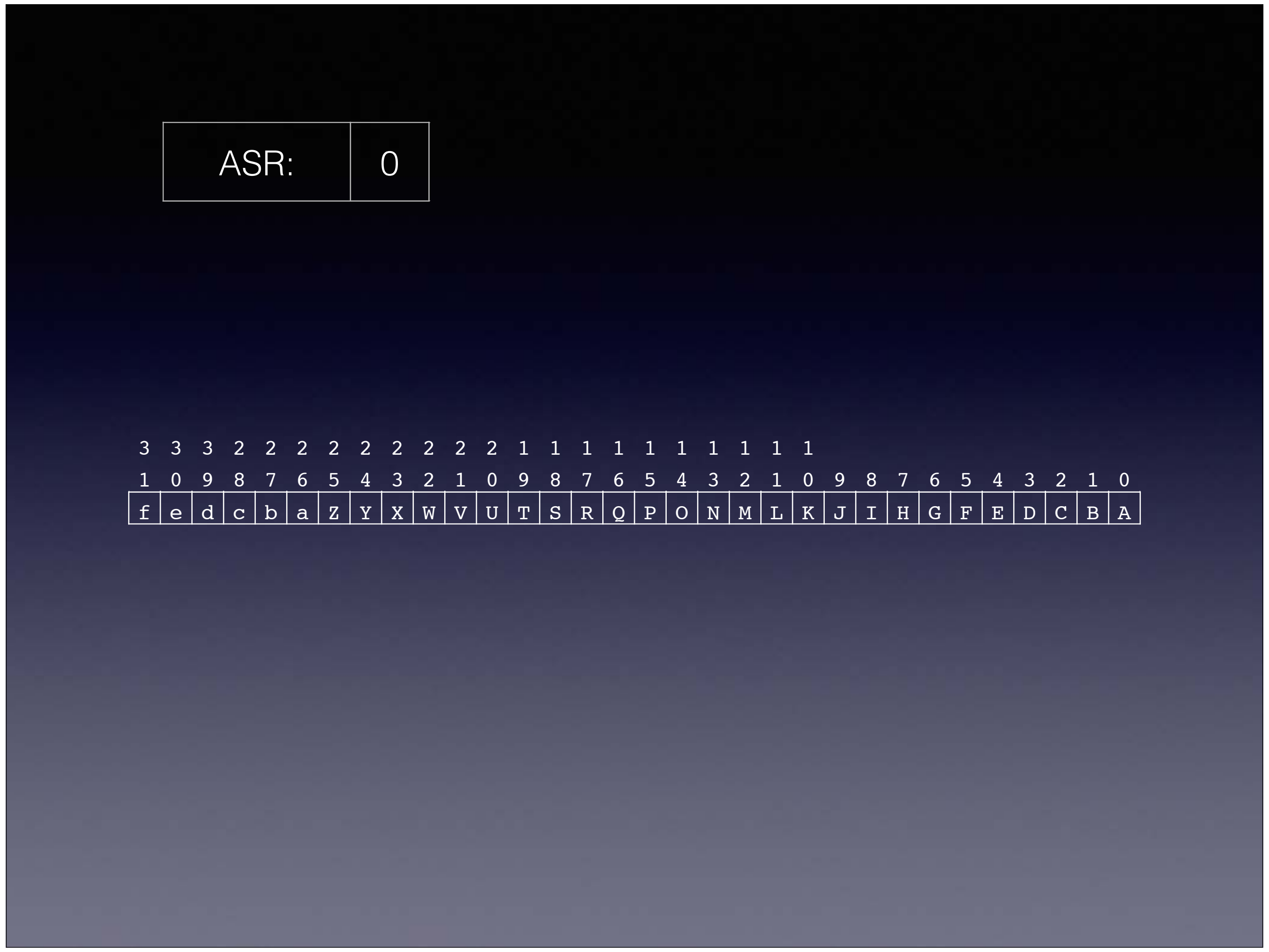
3	3	3	2	2	2	2	2	2	2	2	2	1	1	1	1	1	1	1	1	1	1										
1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0
T	S	R	Q	P	O	N	M	L	K	J	I	H	G	F	E	D	C	B	A	0	0	0	0	0	0	0	0	0	0	0	0

Arithmetic or Logical

- Shifting left is easy, just insert zeros from the right to keep it at 32-bits
- Shifting to the right is more complex
- If we insert zeroes from the left, we'll change the sign
- So there's two types of right shift — arithmetic and logical

Arithmetic or logical

- Logical shifts always shift in a zero bit into the spare bits
- Arithmetic shift right will shift in a copy of the the sign bit (i.e. bit 31) so that the sign of the number doesn’t change



ASR: | 0

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
f	e	d	c	b	a	Z	Y	X	W	V	U	T	S	R	Q	P	O	N	M	L	K	J	I	H	G	F	E	D	C	B	A

Arithmetic Shift Right

ASR:	1

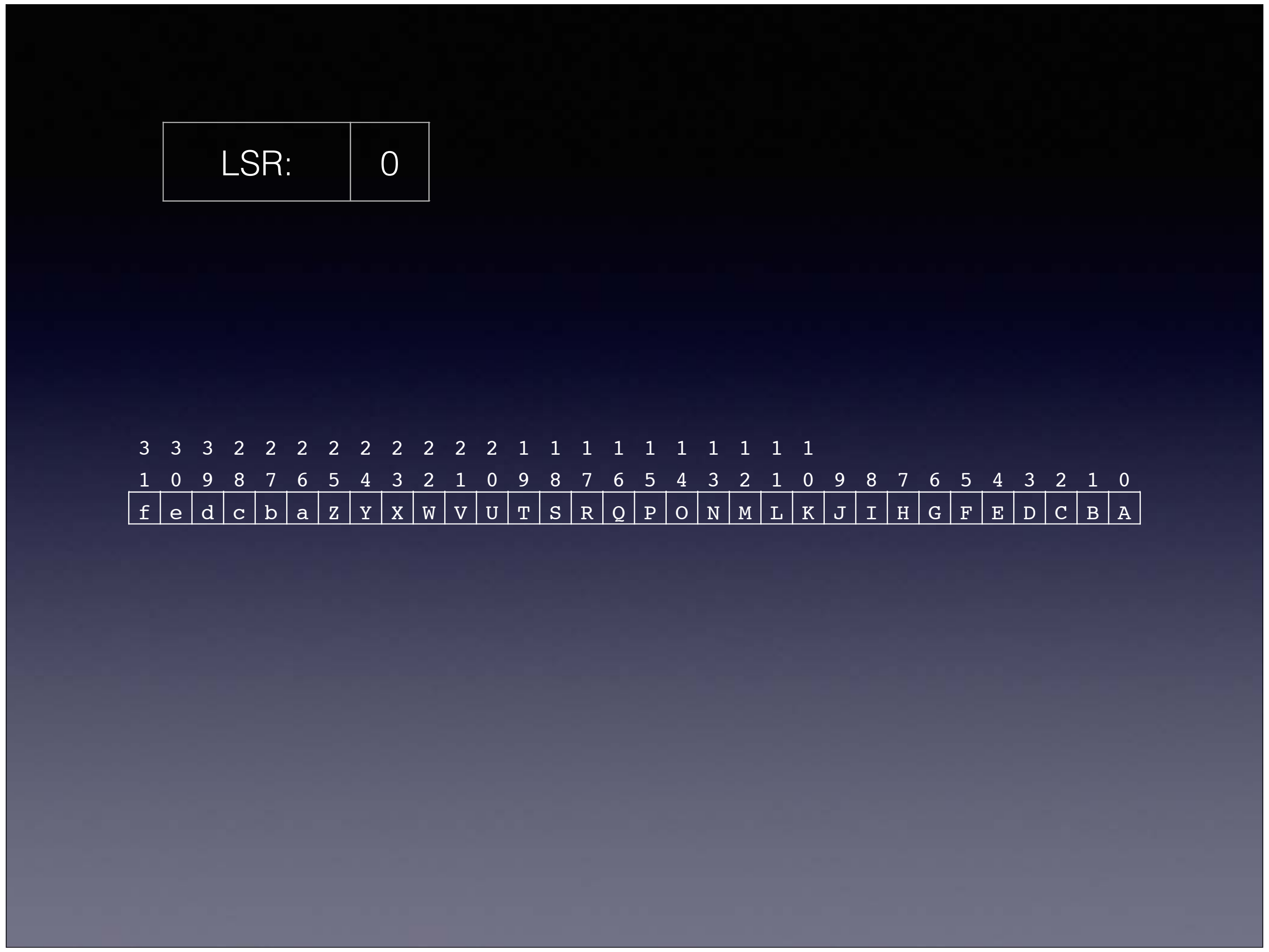
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
f	f	e	d	c	b	a	Z	Y	X	W	V	U	T	S	R	Q	P	O	N	M	L	K	J	I	H	G	F	E	D	C	B

ASR:	2

3	3	3	2	2	2	2	2	2	2	2	2	1	1	1	1	1	1	1	1	1	1										
1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0
f	f	f	e	d	c	b	a	Z	Y	X	W	V	U	T	S	R	Q	P	O	N	M	L	K	J	I	H	G	F	E	D	C

ASR	12

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
f	f	f	f	f	f	f	f	f	f	f	f	e	d	c	b	a	Z	Y	X	W	V	U	T	S	R	Q	P	O	N	M	L



LSR: 0

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
f	e	d	c	b	a	Z	Y	X	W	V	U	T	S	R	Q	P	O	N	M	L	K	J	I	H	G	F	E	D	C	B	A

Logical Shift Right

LSR:	1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	f	e	d	c	b	a	Z	Y	X	W	V	U	T	S	R	Q	P	O	N	M	L	K	J	I	H	G	F	E	D	C	B

ASR:	2

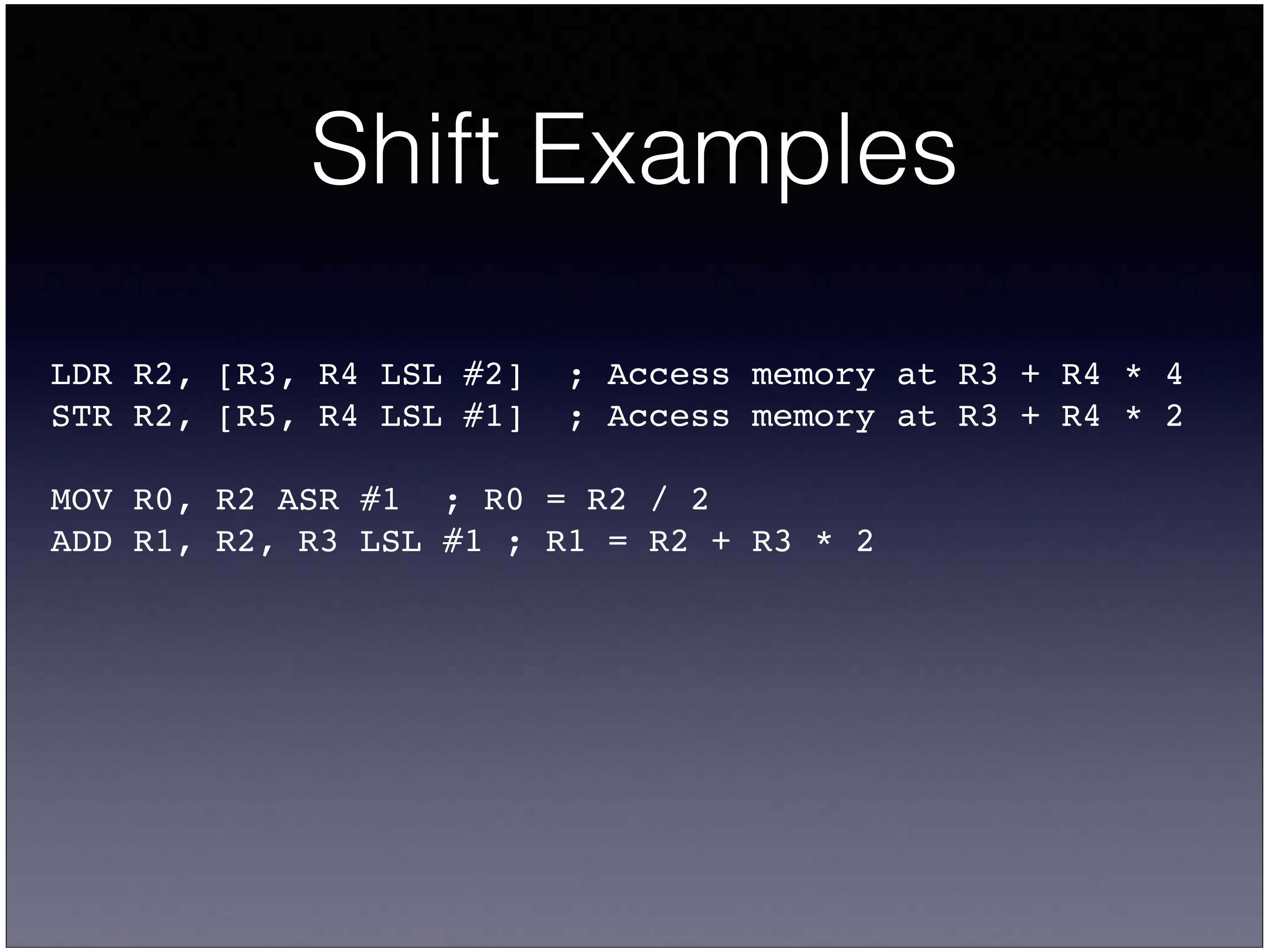
3	3	3	2	2	2	2	2	2	2	2	2	1	1	1	1	1	1	1	1	1	1										
1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0
0	0	f	e	d	c	b	a	Z	Y	X	W	V	U	T	S	R	Q	P	O	N	M	L	K	J	I	H	G	F	E	D	C

ASR	12

3	3	3	2	2	2	2	2	2	2	2	2	1	1	1	1	1	1	1	1	1	1										
1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	f	e	d	c	b	a	Z	Y	X	W	V	U	T	S	R	Q	P	O	N	M	L

Shifts

- ARM supports four types of shift
  - Logical Shift Left
  - Logical Shift Right
  - Arithmetic Shift Right
  - Rotate Right



Shift Examples

```
LDR R2, [R3, R4 LSL #2]  ; Access memory at R3 + R4 * 4
STR R2, [R5, R4 LSL #1]  ; Access memory at R3 + R4 * 2

MOV R0, R2 ASR #1  ; R0 = R2 / 2
ADD R1, R2, R3 LSL #1 ; R1 = R2 + R3 * 2
```

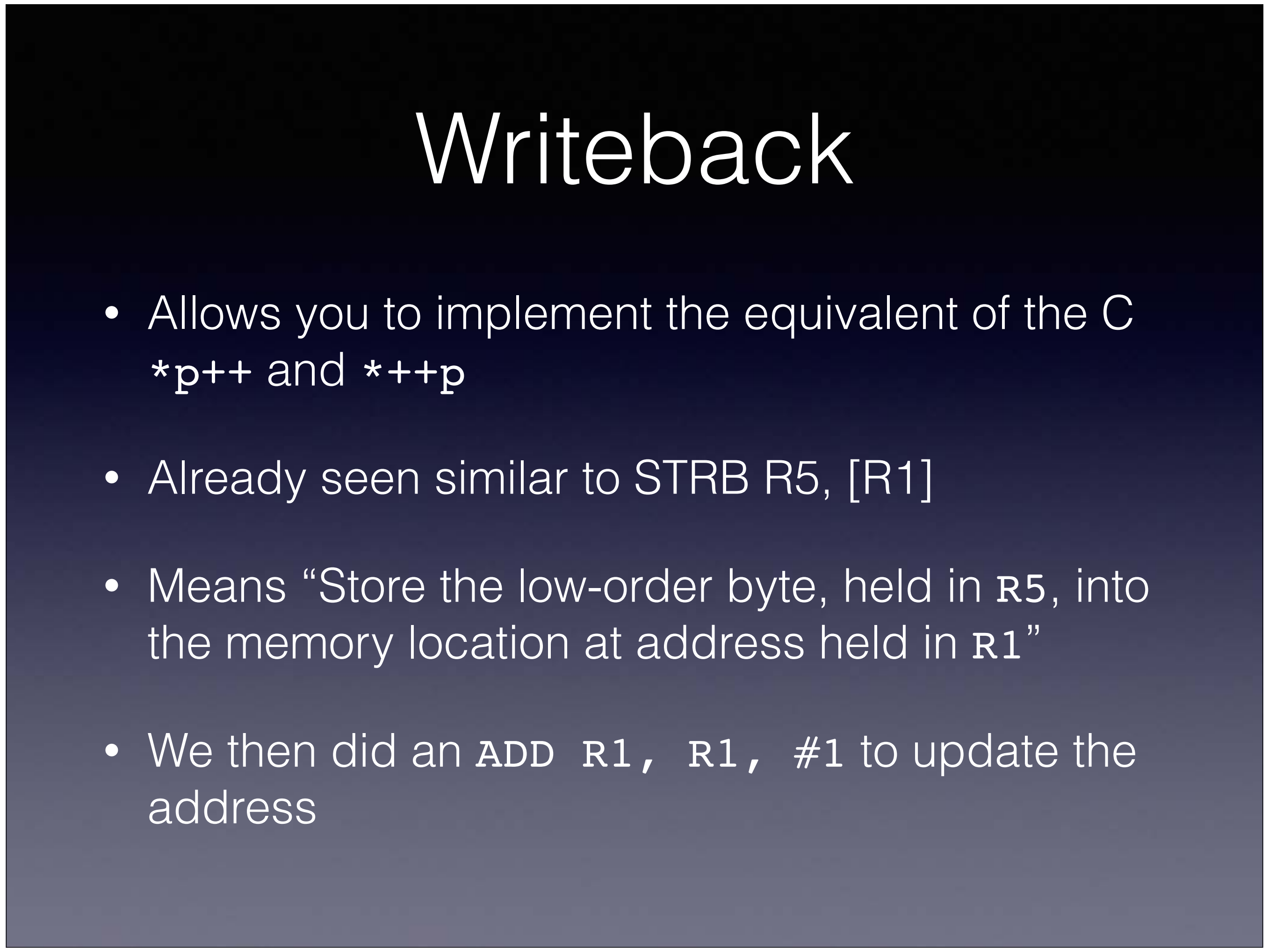
And lots more possible

Top two show how we can make array access very easy

Modify strlen example to use index addressing

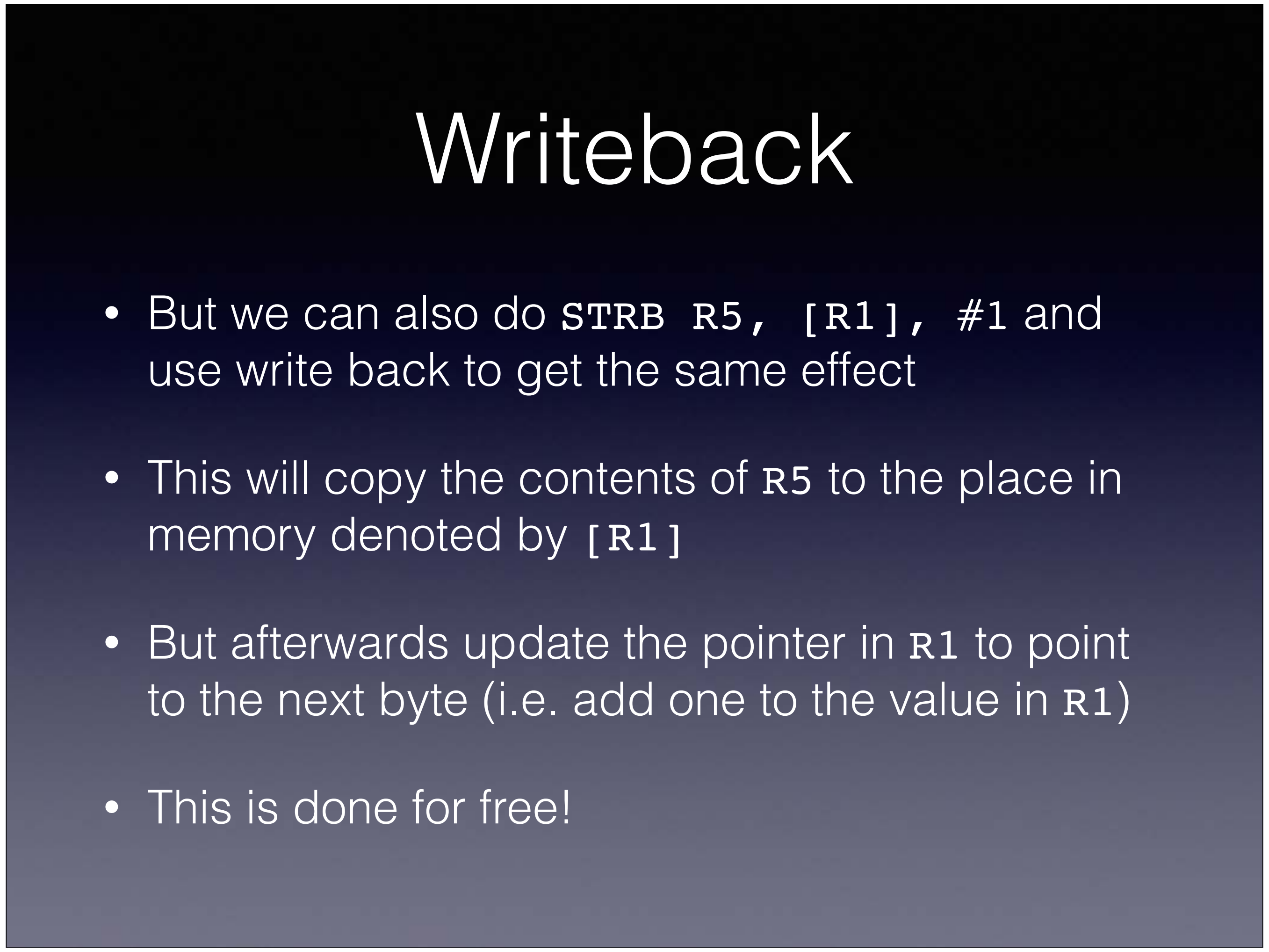
Writeback

- But there’s more…
- ARM’s `LDR` and `STR` instructions also allow you to write the calculated address back into the base register
- This is called *writeback*
- Means that we can auto update the address register for free



Writeback

- Allows you to implement the equivalent of the C `*p++` and `*++p`
- Already seen similar to STRB R5, [R1]
- Means “Store the low-order byte, held in `R5`, into the memory location at address held in `R1`”
- We then did an `ADD R1, R1, #1` to update the address



Writeback

- But we can also do `STRB R5, [R1], #1` and use write back to get the same effect
- This will copy the contents of `R5` to the place in memory denoted by `[R1]`
- But afterwards update the pointer in `R1` to point to the next byte (i.e. add one to the value in `R1`)
- This is done for free!

No extra cycles spent on doing this

Go and show how we can alter strlen.s to use write back and compare the different addresses.

Writeback

- Note the value added on is in bytes
- It is *not* converted to the size of the type as C does
- Need to do this manually (i.e. `[R1],#4`)
- Can also use a register here (with optional shift)

Pre-index Writeback

- Can also do the same with a pre-index operation
- But the syntax is different
- Here we just put an `!` after the square brackets, e.g. `LDRB R1, [R0, #4]!`
- This loads `R1` with the value at memory location `R0+4`
- And updates `R0` to contain `R0 + 4`

ARM Indirect Addressing Summary

- Two possibilities for each of:
  - Address used: Register or Register + offset
  - Final register value: Unchanged or +offset

`Rn` is base register	`Address = Rn`	`Address = Rn + offset`
Rn unchaged	`[Rn]` **indirect** e.g. `[R1]`	`[Rn, offset]` **pre-indexed with offset**, e.g.
Rn = Rn + offset	`[Rn], offset` **post-indexed** e.g. `[R1] , #1`	`[Rn, offset]!` **pre-indexed with write back** e.g. `[R1, #12]!`

ARM Indirect Addressing Summary

- Offset can be a **`number`**, **`-number`**, Register or -Register
- Register offsets can also be shifted…