Module 1-C

Addressing modes and conditionals

Tim Rogers 2022

Reading

This slide set:

- Textbook, Chapter 6, “Branch and Conditional Execution”, pages 111 – 132.
  - Talking about this in this lecture.
- Textbook, Chapter 5, “Load and Store”, pages 97 – 110.
  - Talking about this in the second half of the lecture.

More Reading

End of the week:

- Textbook, Chapter 7,"Structured Programming", pages 133 – 160.
  - We'll talk about this soon.
- Textbook, Chapter 8, "Subroutines", pages 161 – 202.
  - We'll talk about this soon.

- ARMv6-M Architecture Reference Manual (436 pages)
  - Get familiar with sections A5.2 (16-bit Thumb encoding) and A6.7 (Alphabetical list of ARMv6-M Thumb instructions)

Learning Outcome #1

"An ability to program a microcontroller to perform various tasks"

How?

- ✓ A. Architecture and Programming Model
- ✓ B. Instruction Set Basics
- $\boxed{\text{C. Addressing modes and conditionals}}$
- D. Practical Assembly Language Programming
- E. Stacks and Recursive Functions*

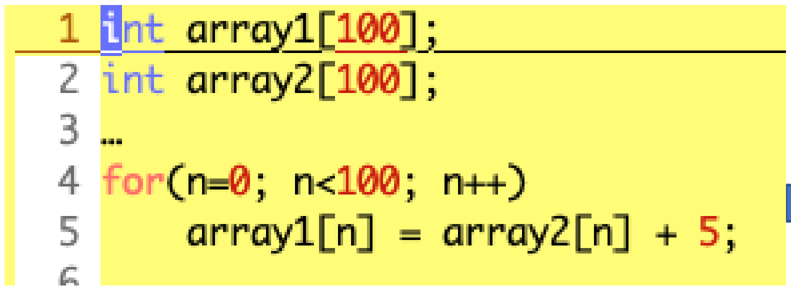
Objective

“Addressing modes and conditionals”

Typically

Why?

High-level lang. code

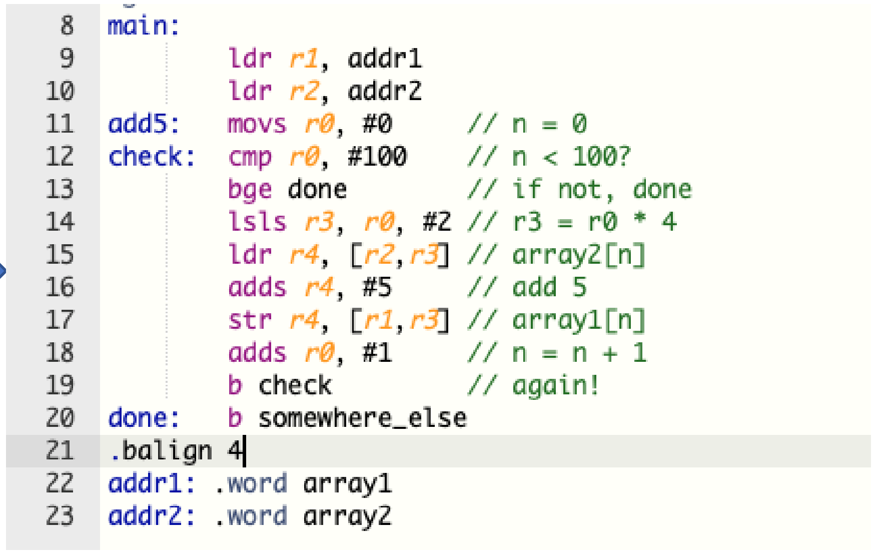


```c
int array1[100];
int array2[100];
…
for(n=0; n<100; n++)
    array1[n] = array2[n] + 5;
```

Compiler

Eg. gcc,vscc, llvm

Assembly Code



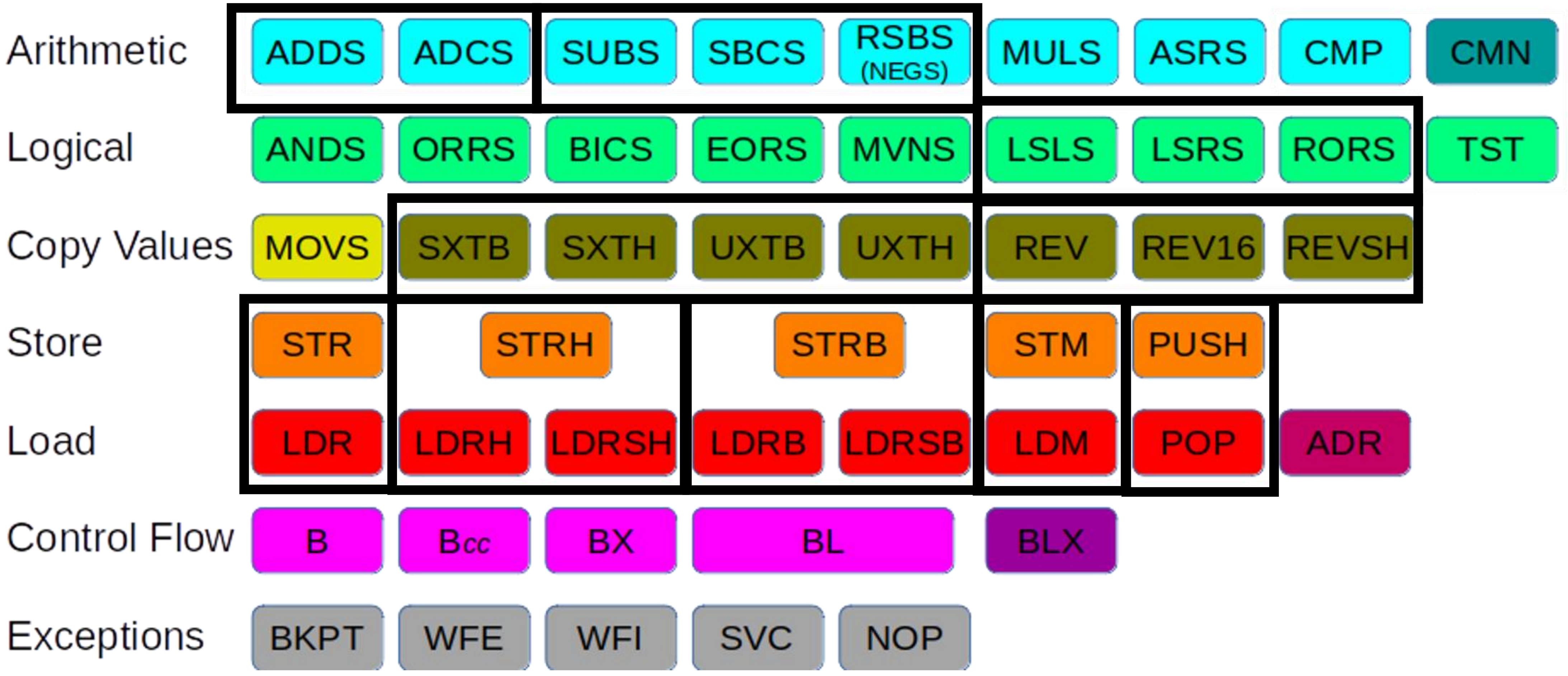
```
main:
        ldr r1, addr1
        ldr r2, addr2
add5:   movs r0, #0       // n = 0
check:  cmp r0, #100      // n < 100?
        bge done          // if not, done
        lsls r3, r0, #2   // r3 = r0 * 4
        ldr r4, [r2,r3]   // array2[n]
        adds r4, #5       // add 5
        str r4, [r1,r3]   // array1[n]
        adds r0, #1       // n = n + 1
        b check           // again!
done:   b somewhere_else
.balign 4
addr1: .word array1
addr2: .word array2
```

Right now

Still useful to think about high-level language constructs: loops, conditionals, etc…

Recall

Rick’s Handy Cortex-M0 Summary



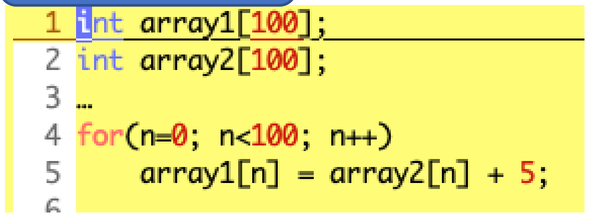
Arithmetic	ADDS	ADCS	SUBS	SBCS	RSBS (NEGS)	MULS	ASRS	CMP	CMN
Logical	ANDS	ORRS	BICS	EORS	MVNS	LSLS	LSRS	RORS	TST
Copy Values	MOVS	SXTB	SXTH	UXTB	UXTH	REV	REV16	REVSH	
Store	STR	STRH		STRB		STM	PUSH		
Load	LDR	LDRH	LDRSH	LDRB	LDRSB	LDM	POP	ADR	
Control Flow	B	Bcc	BX	BL		BLX			
Exceptions	BKPT	WFE	WFI	SVC	NOP				

The Branch

Control Flow | B | Bcc | BX | BL | BLX

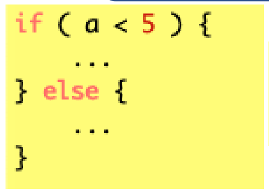
- Most instructions:
  - **PC = PC + sizeOf(Insn) // typically PC = PC + 2**
- Sometimes we don't want that:

Loops

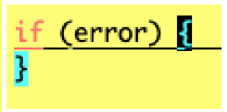


```
int array1[100];
int array2[100];
…
for(n=0; n<100; n++)
    array1[n] = array2[n] + 5;
```

If conditions

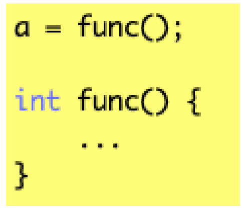


```
if ( a < 5 ) {
    ...
} else {
    ...
}
```



```
if (error) {
}
```

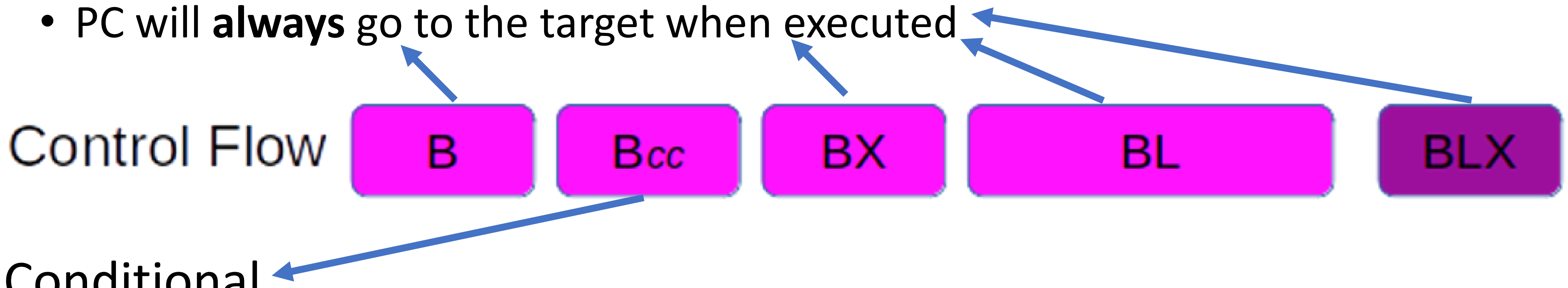
Function Calls



```
a = func();

int func() {
    ...
}
```

Two classes of branches

- Unconditional
  - PC will **always** go to the target when executed



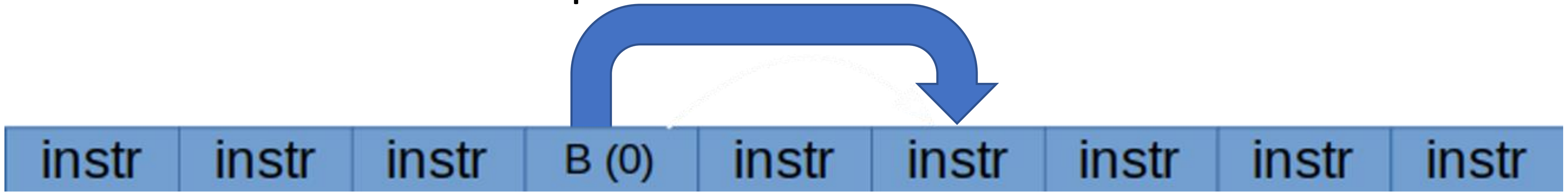
- Conditional
  - PC will go to the target based on a condition

Computing the target: (1) PC-Relative Addressing Mode

- PC Relative Addressing
  - **TargetAddress = PC + SignExt(imm) ***

- * -- Not quite
  - All instructions at lest 2 bytes, always 2-byte aligned.
  - Bottom bits of instruction address always zero
- * -- One more Weirdness…
  - Offsets start 2 instructions past branch.

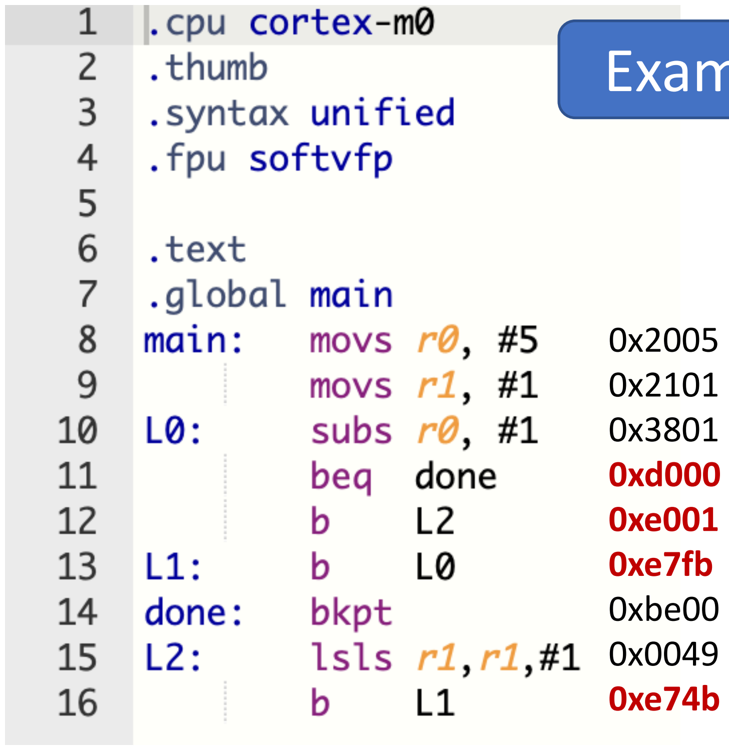
Decimal	Binary
2	1**0**
4	10**0**
1026	10000001**0**



How to use the branch

- When coding: Just use labels ☺
- Thank your local assembler

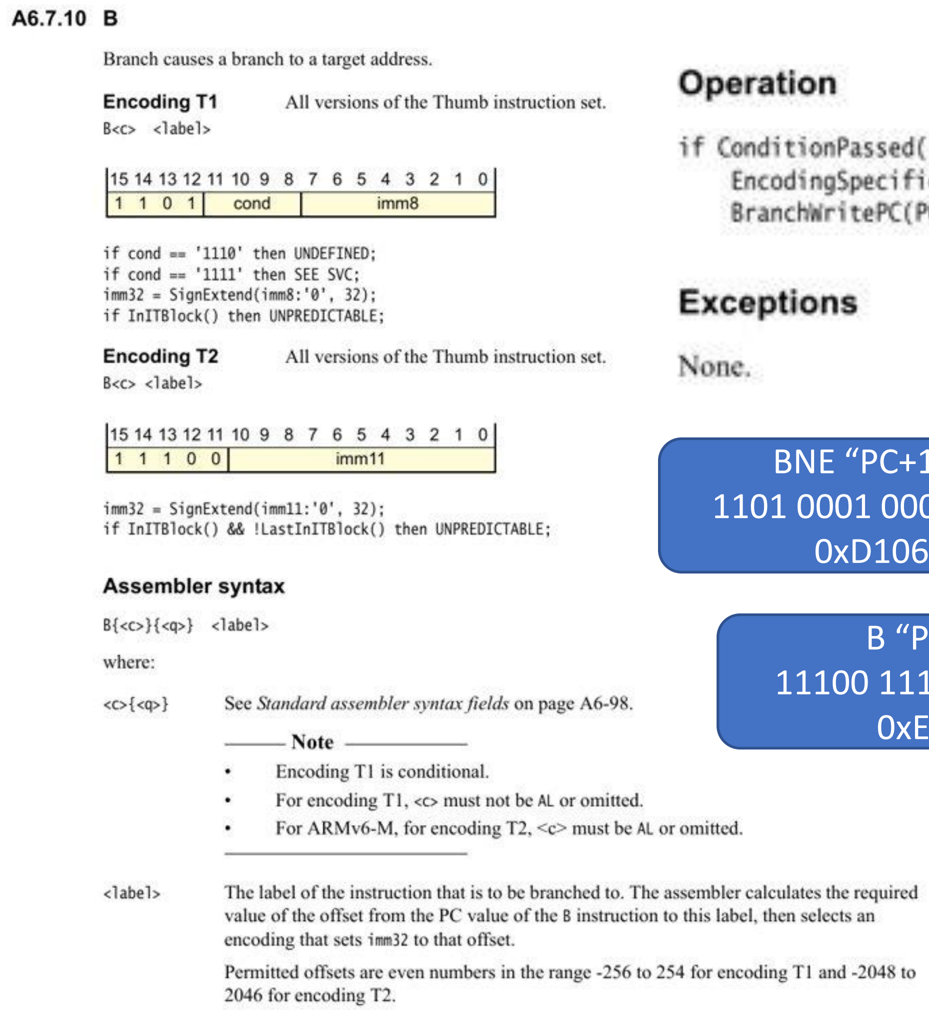
Example 0



```
.cpu cortex-m0
.thumb
.syntax unified
.fpu softvfp

.text
.global main
main:   movs r0, #5     0x2005
        movs r1, #1     0x2101
L0:     subs r0, #1     0x3801
        beq  done       0xd000
        b    L2         0xe001
L1:     b    L0         0xe7fb
done:   bkpt            0xbe00
L2:     lsls r1,r1,#1   0x0049
        b    L1         0xe74b
```

Conditional Branches

- Needed to do anything useful



A6.7.10 B

Branch causes a branch to a target address.

Encoding T1 All versions of the Thumb instruction set.

`B<c> <label>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	cond				imm8							

```
if cond == '1110' then UNDEFINED;
if cond == '1111' then SEE SVC;
imm32 = SignExtend(imm8:'0', 32);
if InITBlock() then UNPREDICTABLE;
```

Encoding T2 All versions of the Thumb instruction set.

`B<c> <label>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	imm11										

```
imm32 = SignExtend(imm11:'0', 32);
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

Assembler syntax

`B{<c>}{<q>} <label>`

where:

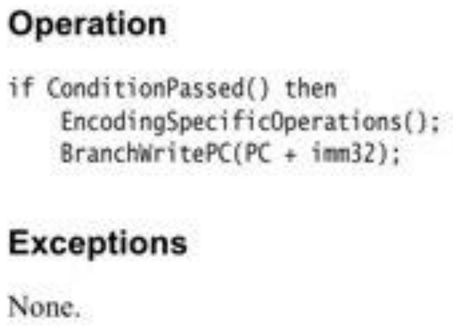
`<c>{<q>}` See *Standard assembler syntax fields* on page A6-98.

Note

- Encoding T1 is conditional.
- For encoding T1, <c> must not be AL or omitted.
- For ARMv6-M, for encoding T2, <c> must be AL or omitted.

`<label>` The label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the B instruction to this label, then selects an encoding that sets imm32 to that offset.

Permitted offsets are even numbers in the range -256 to 254 for encoding T1 and -2048 to 2046 for encoding T2.



Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    BranchWritePC(PC + imm32);
```

Exceptions

None.

BNE "PC+12"
1101 0001 00000110
0xD106

B "PC-2"
11100 11111111111
0xE7FF

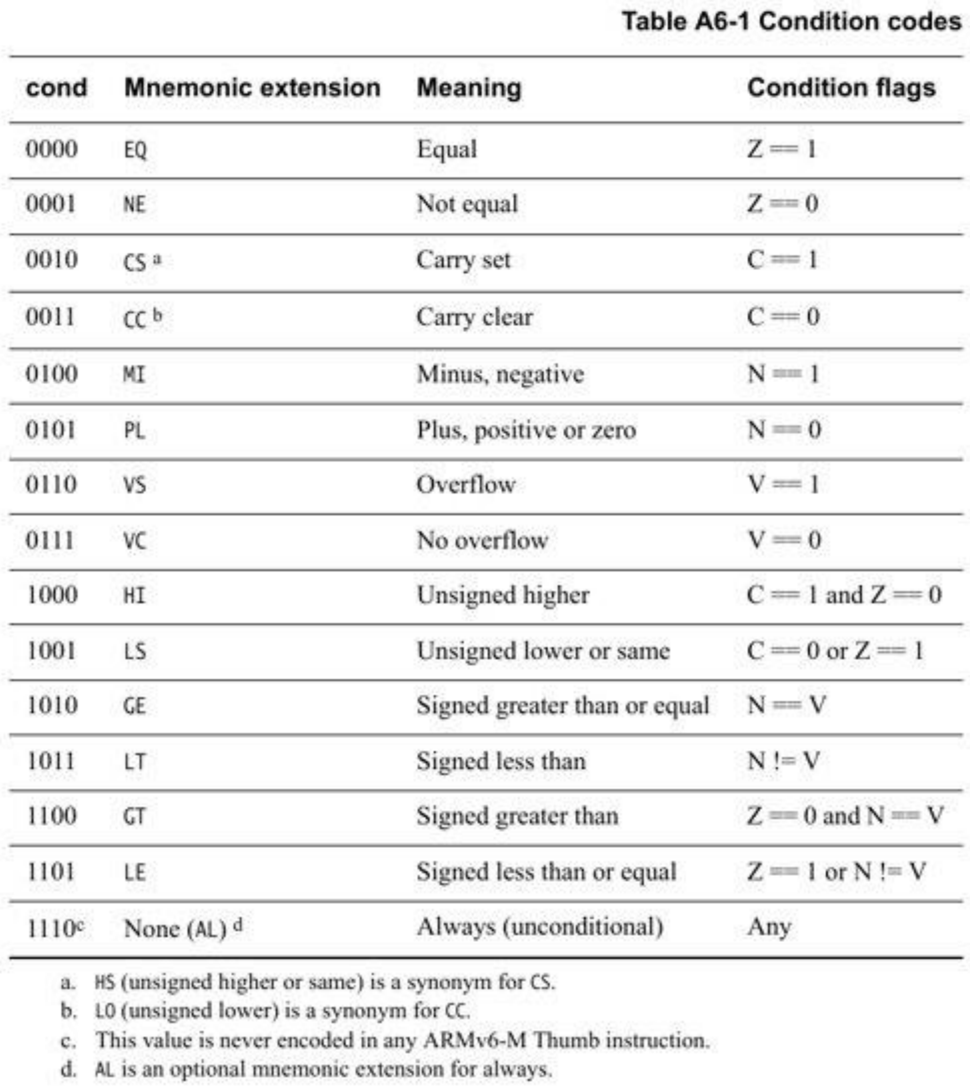


Table A6-1 Condition codes

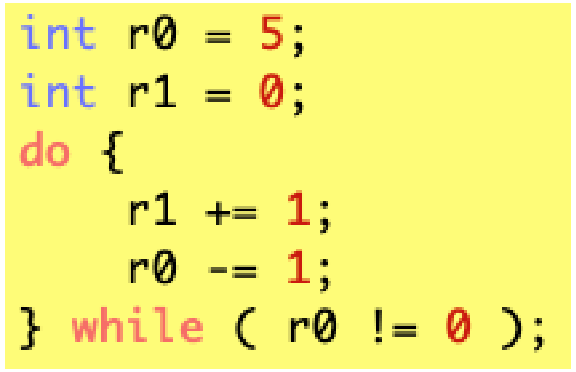
cond	Mnemonic extension	Meaning	Condition flags
0000	EQ	Equal	Z == 1
0001	NE	Not equal	Z == 0
0010	CS [a]	Carry set	C == 1
0011	CC [b]	Carry clear	C == 0
0100	MI	Minus, negative	N == 1
0101	PL	Plus, positive or zero	N == 0
0110	VS	Overflow	V == 1
0111	VC	No overflow	V == 0
1000	HI	Unsigned higher	C == 1 and Z == 0
1001	LS	Unsigned lower or same	C == 0 or Z == 1
1010	GE	Signed greater than or equal	N == V
1011	LT	Signed less than	N != V
1100	GT	Signed greater than	Z == 0 and N == V
1101	LE	Signed less than or equal	Z == 1 or N != V
1110[c]	None (AL) [d]	Always (unconditional)	Any

a. HS (unsigned higher or same) is a synonym for CS.
b. LO (unsigned lower) is a synonym for CC.
c. This value is never encoded in any ARMv6-M Thumb instruction.
d. AL is an optional mnemonic extension for always.

A Basic Conversion

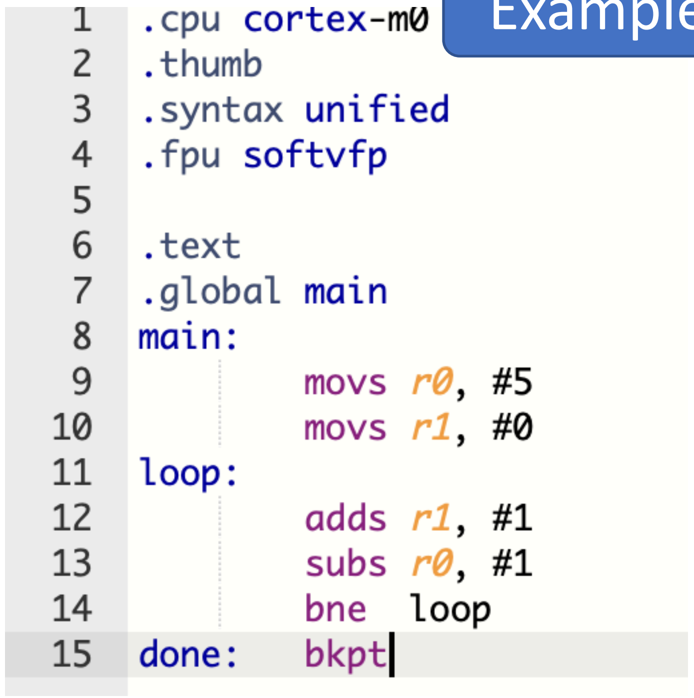
- First loop conversion: C do-while loop
  - Simpler in assembly than while: always do it at least once

C “do-while“ loop



```c
int r0 = 5;
int r1 = 0;
do {
    r1 += 1;
    r0 -= 1;
} while ( r0 != 0 );
```

Example 1

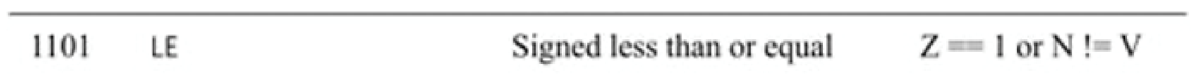


```
.cpu cortex-m0
.thumb
.syntax unified
.fpu softvfp

.text
.global main
main:
        movs r0, #5
        movs r1, #0
loop:
        adds r1, #1
        subs r0, #1
        bne  loop
done:   bkpt
```

More “Exotic” conditions?

- Generally, execute an instruction that mods flags “S” suffix
  - Then do Branch
  - Mnemonics all assume a subtraction right before

```
SUBS    R6, R2, R5  // R2 – R5
BLE     error       // Branch if R2 is <= R5 (sign)
```



1101	LE	Signed less than or equal	Z == 1 or N != V

Signed vs. Unsigned

- This only matters when you are working with very large integers. For instance: 0xffffffff
  - Is this a negative number?
    - It depends on whether we choose to interpret it as a signed or unsigned number.
    - If we want to treat it as a signed integer, it is -1
    - If we want to treat it as unsigned, it is 4,294,967,295

Unsigned Integers

- Recall:
  - n bits give rise to 2^n combinations
  - let us call a string of 32 bits as “$b_{31}\ b_{30} \ldots b_3\ b_2\ b_1\ b_0$”
- $f(b_{31} \ldots b_0) = b_{31} \times 2^{31} + \ldots + b_1 \times 2 + b_0 \times 2^0$
- Treat as normal binary number
  - e.g., 0 . . .011010101

  $= 1 \times 2^7 + 1 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$

  $= 128 + 64 + 16 + 4 + 1 = 213$
- max $f(111 \ldots 11) = 2^{32} - 1 =$ 4, 294, 967, 295
- min $f(000 \ldots 00) = 0$
- range $[0, 2^{32}-1]$ => # values $(2^{32} - 1) - 0 + 1 = 2^{32}$

Signed Integers

- 2's complement
- $f(b_{31}\ b_{30} \ldots b_1\ b_0) = -b_{31} \times 2^{31} + \ldots + b_1 \times 2 + b_0 \times 2^0$
  - max $f(0111 \ldots 11) = 2^{31} - 1 = 2147483647$
  - min $f(100 \ldots 00) = -2^{31} = -2147483648$ (asymmetric)
- range $[-2^{31}, 2^{31}-1]$ => #values $(2^{31}-1 - -2^{31} + 1) = 2^{32}$
- E.g., -6
- 000 . . . 0110 --> 111 . . 1001 + 1 --> 111 . . .1010

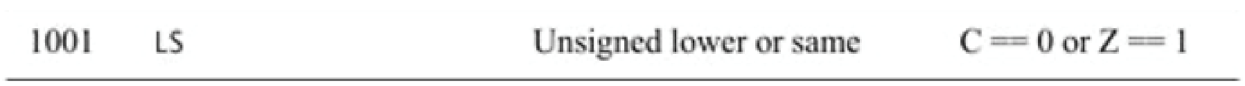
Signed or unsigned data types

- In C, we can explicitly declare variables that are never negative (always positive or zero) like this:
  - unsigned int   big32 = 0xffffffff; // AKA uint32_t
  - unsigned short big16 = 0xffff;   // AKA uint16_t
  - unsigned char big8 = 0xff;   // AKA uint8_t
- We'll look at this again when we talk about converting C into assembly language.

Back to branches: unsigned values

- Subtraction operation is the same, regardless of sign
  - Beauty of 2's compliment!
- If we know that R2 and R5 are unsigned, we just interpret the flag differently
  - BLE: Signed, BLS: Unsigned

```
SUBS    R6, R2, R5  // R2 – R5
BLS     error       // Branch if R2 is <= R5 (unsign)
```

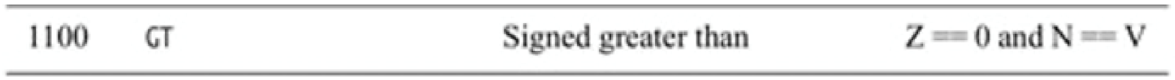


1001	LS	Unsigned lower or same	C == 0 or Z == 1

Avoiding the subtraction

- Can use the CMP instruction
  - Sets flags based on the subtraction.
  - Code branches to error if R2 > R5
  - After CMP insn R2 is unchanged.
- No “S” on CMP, because it’s only used to set flags – always sets them.

```
CMP     R2, R5     // R2 – R5: set flags, don’t store result
BGT     error      // Branch if R2 > R5 (sign)
```

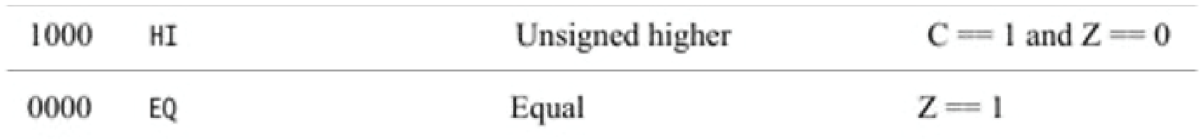


1100	GT	Signed greater than	Z == 0 and N == V

Unsigned greater than or equal to

- No single instruction for unsigned gte
- Can do it with 2 branch instructions
  - Only need 1 CMP
  - Branch instructions don't edit flags – can reuse results.

```
CMP      R2, R5     // R2 – R5: set flags, don't store result
BHI      error      // Branch if R2 is > R5 (unsign)
BEQ      error      // Branch if R2 equal to R5
```



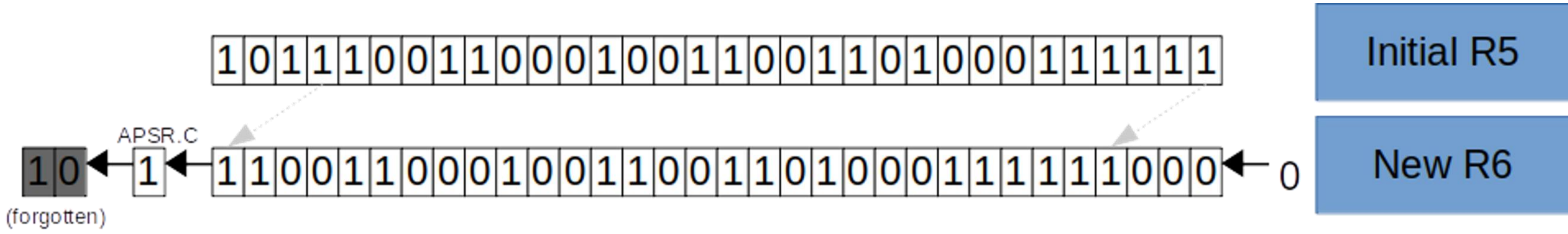
1000	HI	Unsigned higher	C == 1 and Z == 0
0000	EQ	Equal	Z == 1

Recall

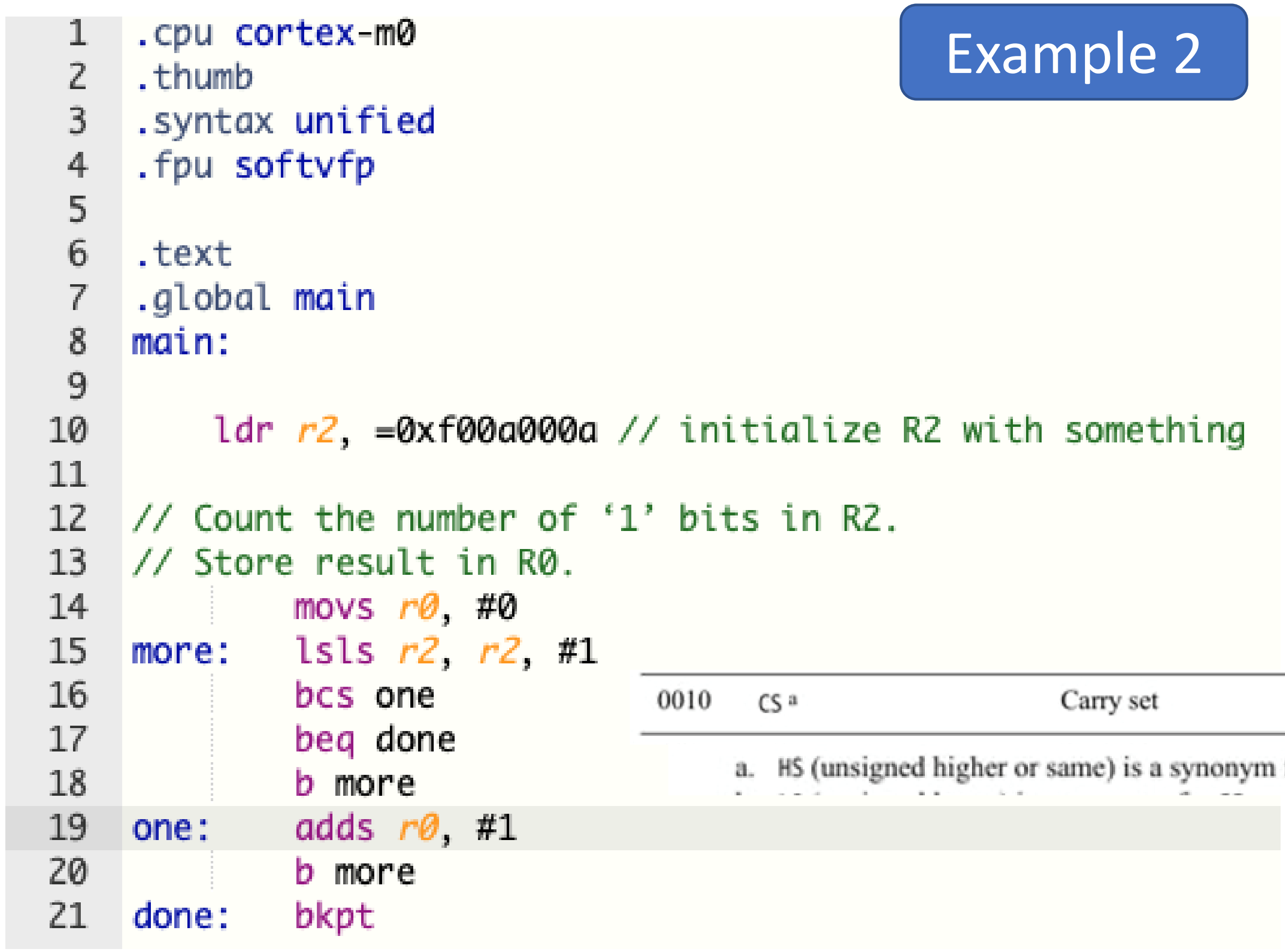
Logical Shift Left

- Can shift the values in 32-bit registers
  - << operator in C
  - Last bit shifted out into APSR.C
  - Zeros shifted in

LSLS R6, R5, #3

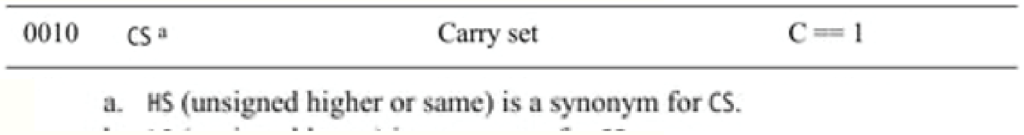


Can use LSL to test bits



Example 2

```
.cpu cortex-m0
.thumb
.syntax unified
.fpu softvfp

.text
.global main
main:

    ldr r2, =0xf00a000a // initialize R2 with something

// Count the number of ‘1’ bits in R2.
// Store result in R0.
        movs r0, #0
more:   lsls r2, r2, #1
        bcs one
        beq done
        b more
one:    adds r0, #1
        b more
done:   bkpt
```

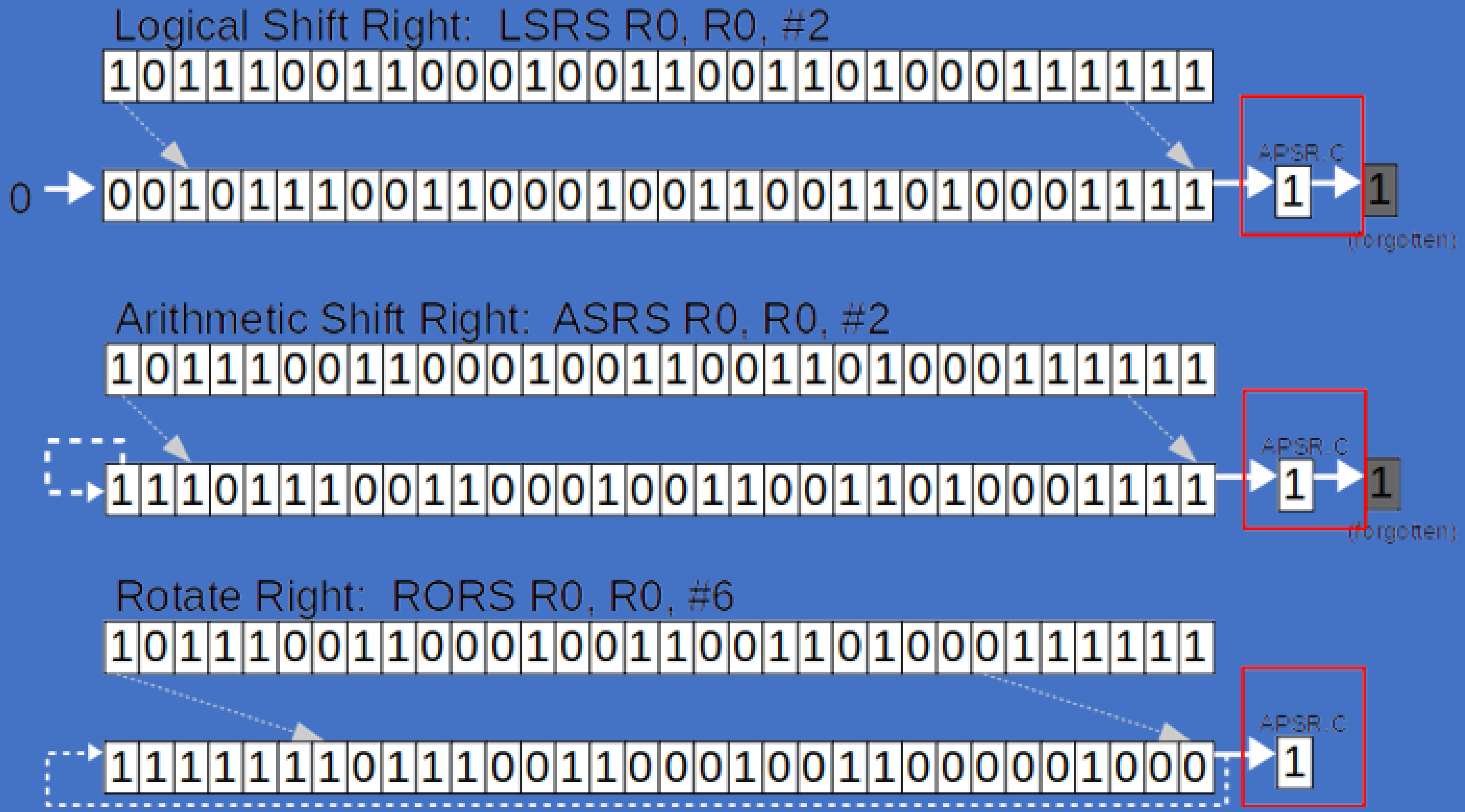


0010	CS [a]	Carry set	C == 1

a. HS (unsigned higher or same) is a synonym for CS.

- Leftmost bit in the carry flag
- BCS: Branch if carry set
  - Can also use this instead of BHI+BEQ from 2 slides ago

Shift and Rotate



Initial R0

New R0

Initial R0

New R0

Initial R0

New R0

Bitwise logical instructions

ANDS R0, R1

```
1 0 1 1 1 0 0 1 1 0 0 0 1 0 0 1 1 0 0 1 1 0 1 0 0 0 1 1 1 1 1 1
0 0 0 0 1 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 0 0 0 0 0 1 1 1 1 0 0   (mask)
0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 1 1 0 0 1 1 0 0 0 0 0 1 1 1 1 0 0
```

Initial R0
AND
R1
New R0

ORRS R0, R1

```
1 0 1 1 1 0 0 1 1 0 0 0 1 0 0 1 1 0 0 1 1 0 1 0 0 0 1 1 1 1 1 1
0 0 0 0 1 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 0 0 0 0 0 1 1 1 1 0 0   (bits to set)
1 0 1 1 1 0 0 1 1 0 0 0 1 1 1 1 1 1 1 1 1 0 1 0 0 0 1 1 1 1 1 1
```

Initial R0
OR
R1
New R0

BICS R0, R1

```
1 0 1 1 1 0 0 1 1 0 0 0 1 0 0 1 1 0 0 1 1 0 1 0 0 0 1 1 1 1 1 1
0 0 0 0 1 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 0 0 0 0 0 1 1 1 1 0 0   (bits to clear)
1 0 1 1 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 1
```

Initial R0
AND NOT
R1
New R0

Bitwise logical instructions

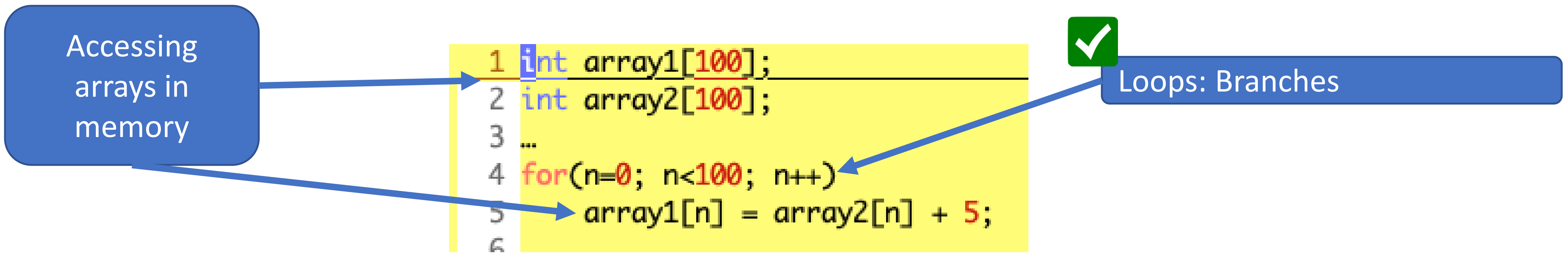
EORS R0, R1

1 0 1 1 1 0 0 1 1 0 0 0 1 0 0 1 1 0 0 1 1 0 1 0 0 0 1 1 1 1 1 1	Initial R0 AND R1
0 0 0 0 1 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 0 0 0 0 0 1 1 1 1 0 0 (bits to toggle)	
1 0 1 1 0 0 0 1 1 0 0 0 0 1 1 0 0 1 1 0 0 0 1 0 0 0 0 0 0 0 1 1	New R0

MVNS R0, R1 // Also known as NOT

1 0 1 1 1 0 0 1 1 0 0 0 1 0 0 1 1 0 0 1 1 0 1 0 0 0 1 1 1 1 1 1	Initial R1
0 1 0 0 0 1 1 0 0 1 1 1 0 1 1 0 0 1 1 0 0 1 0 1 1 1 0 0 0 0 0 0	New R0

Next Piece: Need to better understand memory



```
int array1[100];
int array2[100];
…
for(n=0; n<100; n++)
    array1[n] = array2[n] + 5;
```

Recall: Our addressing modes

- ✓ **Immediate**: small value contained in instruction
- ✓ **Register**: src/dst register in instruction
- **Offset**: value is at an address that's a relative (constant) offset from a register
- **Indexed**: value is at address specified by register plus a second register

Recall

The LDR (Load Register) Instruction

- Let’s say R0 holds the value 0x20000000.
- We want to load the four-byte value contained in addresses
- 0x20000000 – 0x20000003 into register R1.
- This is the instruction:
  - LDR R1, [R0]
- “Use the value in R0 as an address. Read 4-bytes starting from that address, and put the result in R1.”
- Caveat: R0 must be evenly divisible by 4.
  - If not, it will invoke a fault handler. You don’t want that.

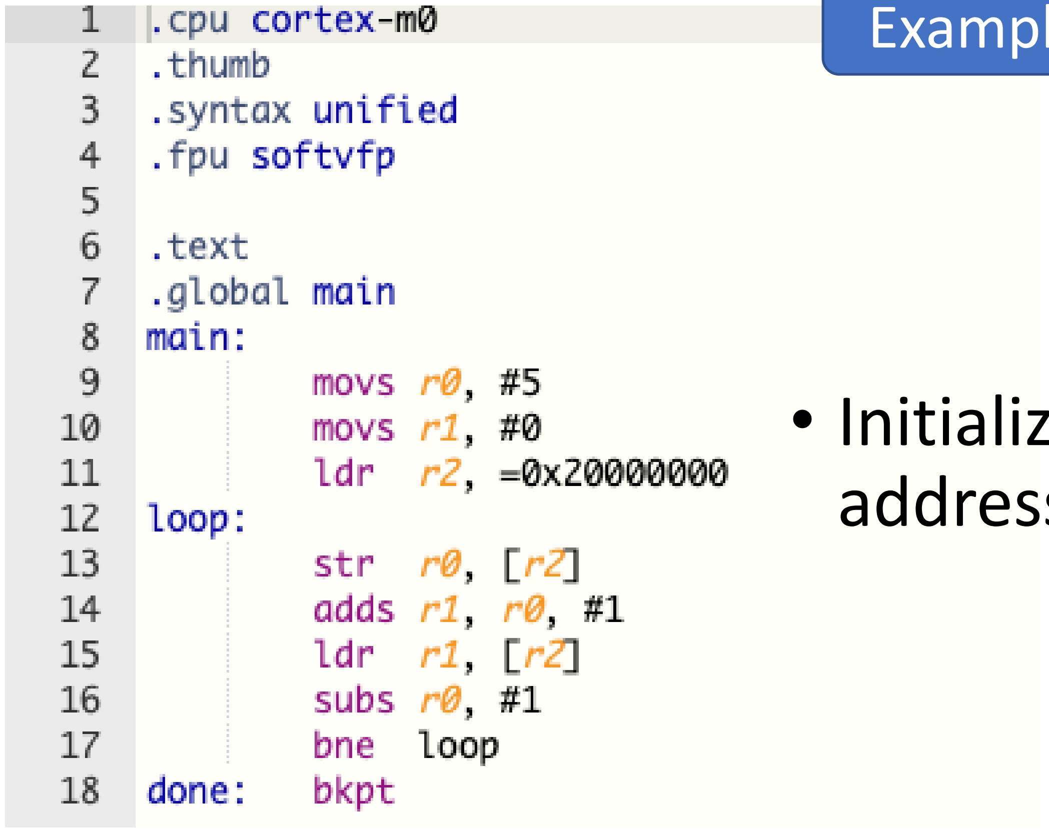
Recall

The STR (Store Register) Instruction

- Let’s say R0 holds the value 0x20000000.
- We want to store the four-byte value contained in R1 to the addresses 0x20000000 – 0x20000003.
- This is the instruction:
  - STR R1, [R0]
- “Use the value in R0 as an address. Write the value of R1 into the 4-bytes starting at that address.”
- Caveat: R0 must be evenly divisible by 4.
  - If not, it will invoke a fault handler. You don’t want that.

Simple Store/Load Example

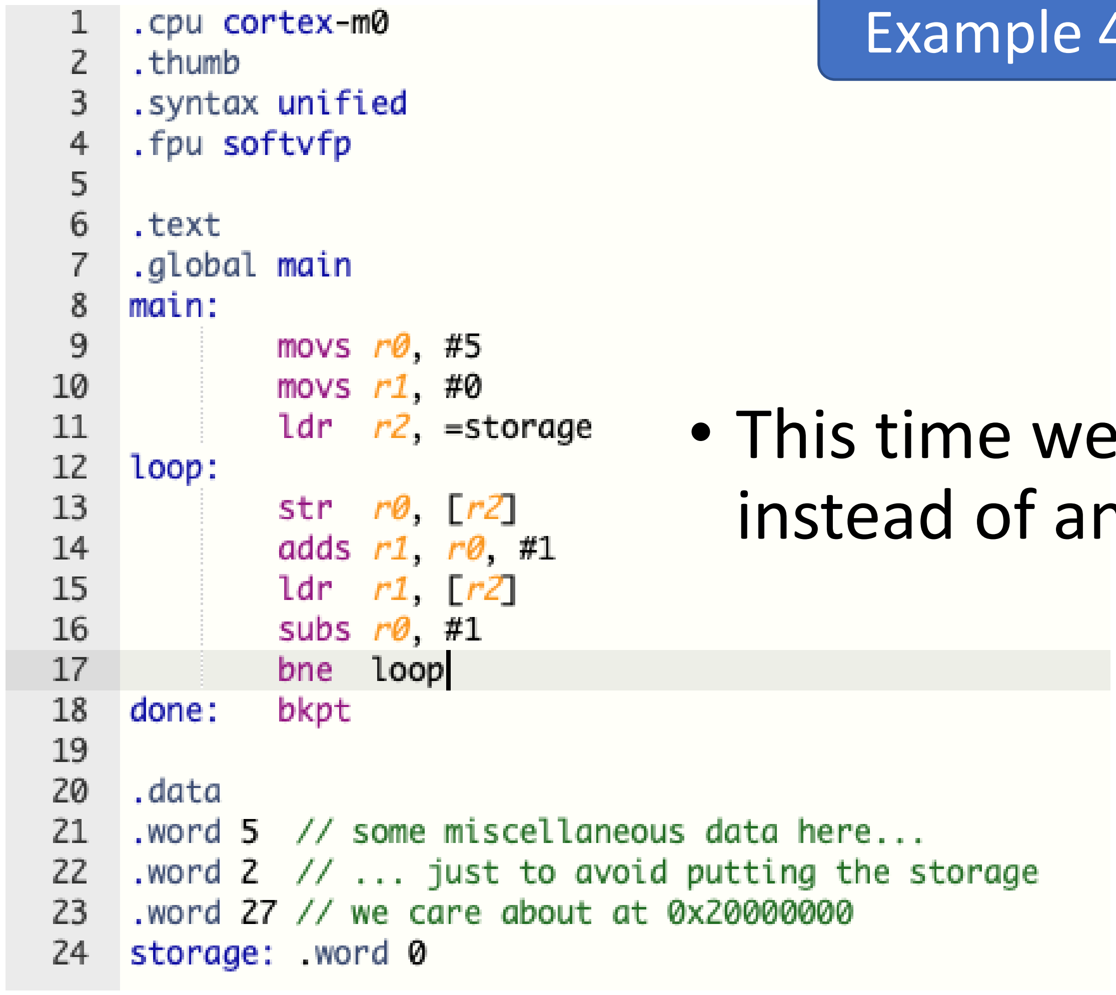
Example 3



```
.cpu cortex-m0
.thumb
.syntax unified
.fpu softvfp

.text
.global main
main:
        movs r0, #5
        movs r1, #0
        ldr  r2, =0x20000000
loop:
        str  r0, [r2]
        adds r1, r0, #1
        ldr  r1, [r2]
        subs r0, #1
        bne  loop
done:   bkpt
```

- Initializing R2 to an explicit address

Another simple example



```
.cpu cortex-m0
.thumb
.syntax unified
.fpu softvfp

.text
.global main
main:
        movs r0, #5
        movs r1, #0
        ldr  r2, =storage
loop:
        str  r0, [r2]
        adds r1, r0, #1
        ldr  r1, [r2]
        subs r0, #1
        bne  loop
done:   bkpt

.data
.word 5  // some miscellaneous data here...
.word 2  // ... just to avoid putting the storage
.word 27 // we care about at 0x20000000
storage: .word 0
```

Example 4

- This time we used a label, instead of an absolute address

Offset Addressing

- All these LDR/STRs are using ***offset* addressing** with an offset of zero.
- Generally:
  - **LDR/STR Ry, [Rx, #<someConstant>]**
  - i.e.: LDR R1, [R0, #4] – “*Take the value of R0, add 4 to it, use the sum as an address, go to that address, read four contiguous bytes, and put it in R1*”
  - For LDR, can use offset from 0-124
  - **Caveat: R0+offset must be evenly divisible by 4.**

Common use of offset addressing

- Structs in C

```c
struct Position {
    int x;
    int y;
    int z;
};

Position pos;
```

```
// Assume R5 holds address of pos
LDR         R0, [R5, #0]         // Load pos.x
LDR         R1, [R5, #4]         // Load pos.y
LDR         R1, [R5, #8]         // Load pos.z
```

Recall: Our addressing modes

✓ • <u>**Immediate**</u>: small value contained in instruction

✓ • <u>**Register**</u>: src/dst register in instruction

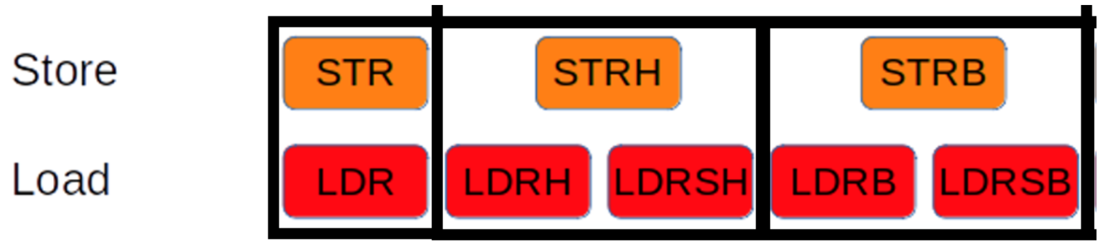
✓ • <u>**Offset**</u>: value is at an address that’s a relative (constant) offset from a register

• <u>**Indexed**</u>: value is at address specified by register plus a second register

Indexed Addressing

- LDR/STR R1, [R0, R2]
- “Take the value of R0, add it to the value of R2, use the sum as an address, read four contiguous bytes, and place them in R1.”
- Similar specification for STR.
- **Caveat: R0+R2 must be evenly divisible by 4.**

Different data sizes

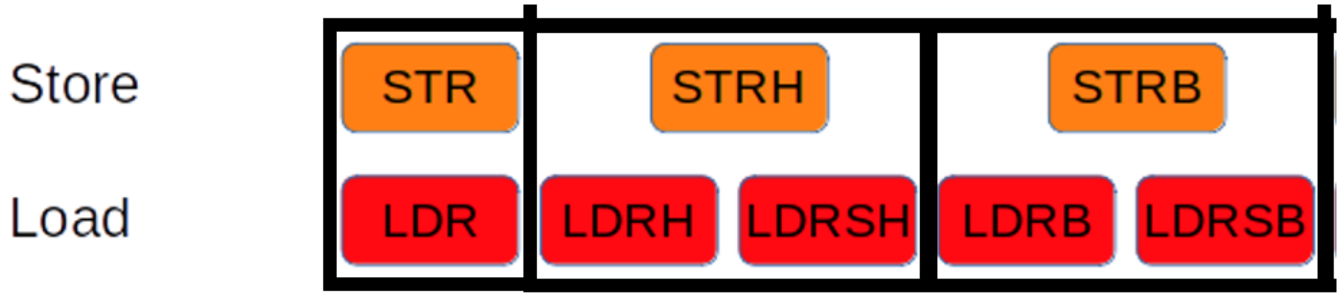


- There are also
  - STRH: store a halfword (2 bytes) [index is a reg or an #imm multiple of 2 from 0 – 62]
  - LDRH: load an unsigned halfword (2 bytes) [index is the same as STRH]
  - LDRSH: load a signed halfword (sign-extend bit 15) [index must be a reg]
  - **Caveat: Halfword load/stores must use an address that is evenly divided by 2.**

  - STRB: store a byte [index is a reg or #imm from 0 – 31]
  - LDRB: load an unsigned byte [index is the same as STRB]
  - LDRSB: load a signed byte (sign-extend bit 7) [index must be a reg]
  - A byte load or store has no alignment requirements.

Why 2 load flavors (signed and unsigned)

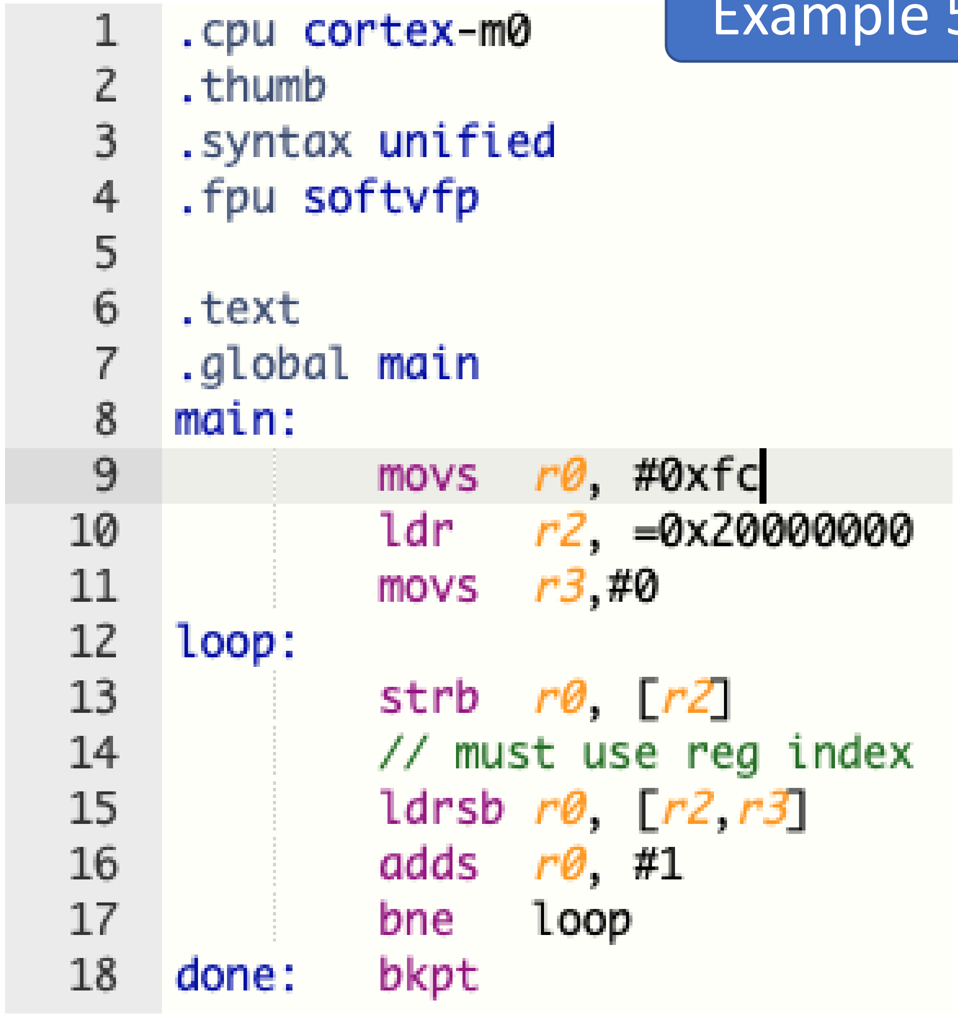
- Only one store type per size
- Because registers are always 32-bits. Operations always on 32-bits.



- When loading less than 32-bits from memory – need to decide how to fill the register
  - i.e. If we are loading a byte (0xff) from memory LDRB (unsigned) puts 0x000000ff in register, LDRSB (signed) puts 0xffffffff.
- Storing – always takes the lowest bits (16 for half, 8 for byte)

Load/Store Bytes

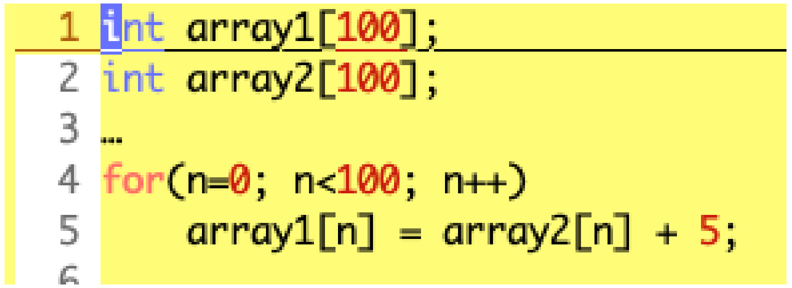
Example 5



```
.cpu cortex-m0
.thumb
.syntax unified
.fpu softvfp

.text
.global main
main:
        movs    r0, #0xfc
        ldr     r2, =0x20000000
        movs    r3,#0
loop:
        strb    r0, [r2]
        // must use reg index
        ldrsb   r0, [r2,r3]
        adds    r0, #1
        bne     loop
done:   bkpt
```

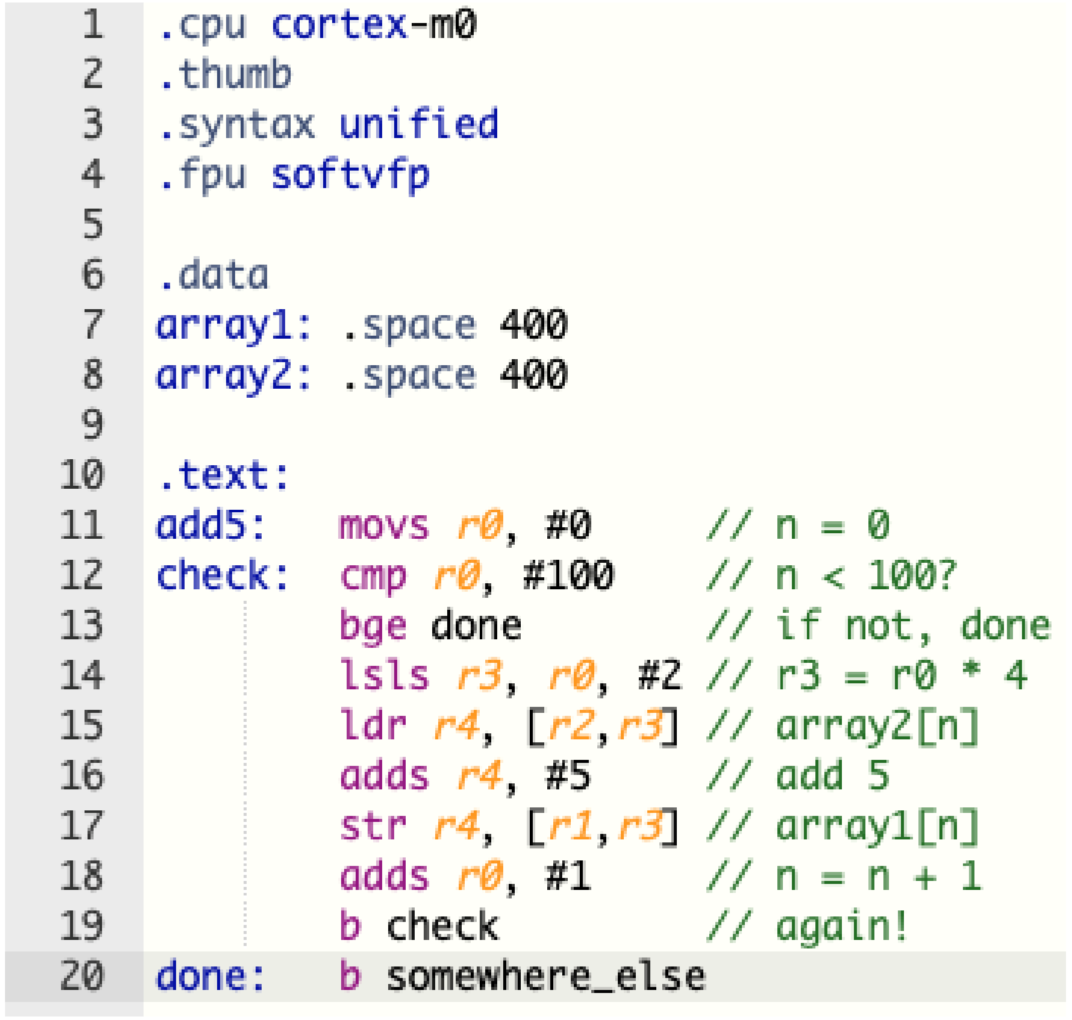
- R0 starts out as 0xfc
- One byte stored in memory
- Loaded from memory as a signed byte, then incremented

Putting it all together



```c
int array1[100];
int array2[100];
…
for(n=0; n<100; n++)
    array1[n] = array2[n] + 5;
```

For now assume:
"n" is in R0
R1 is the address for array1
R2 is the address for array2



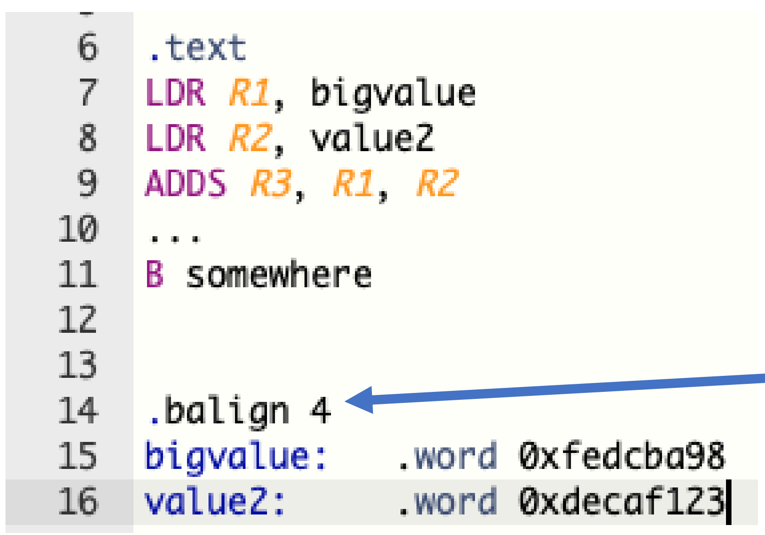
```
.cpu cortex-m0
.thumb
.syntax unified
.fpu softvfp

.data
array1: .space 400
array2: .space 400

.text:
add5:   movs r0, #0       // n = 0
check:  cmp r0, #100      // n < 100?
        bge done          // if not, done
        lsls r3, r0, #2   // r3 = r0 * 4
        ldr r4, [r2,r3]   // array2[n]
        adds r4, #5       // add 5
        str r4, [r1,r3]   // array1[n]
        adds r0, #1       // n = n + 1
        b check           // again!
done:   b somewhere_else
```

Last Piece: Getting values in memory

- We can load things relative to the PC (i.e. LDR R1, [PC, #12])
  - Cannot perform store instructions relative to a PC – instructions are in read-only memory – range not far enough.
- So we can load something from a label LDR R1, label
  - Called a literal load
- **Caveat: The item must be no further than 1020 bytes beyond the PC in the text segment.**

Bonus: PC (rounded up) +12 is automatically evenly divisible by 4

Literal load example



```
.text
LDR R1, bigvalue
LDR R2, value2
ADDS R3, R1, R2
...
B somewhere


.balign 4
bigvalue:    .word 0xfedcba98
value2:      .word 0xdecaf123
```

- Max offset: 1020 bytes
- Words must be in the text segment

Need to make sure the words are aligned.

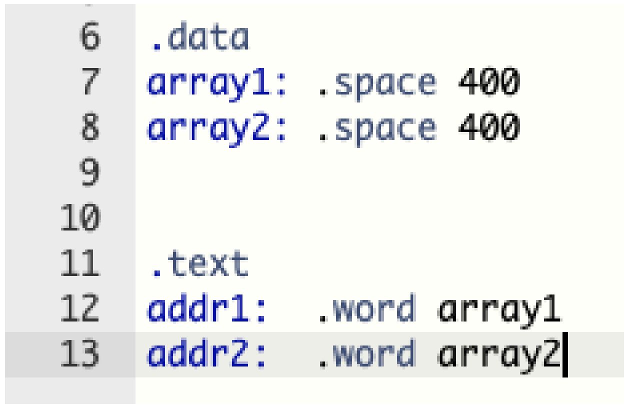
A Label is an address

What value is put into the memory word allocated for addr1?
A: 400
B: 100
C: 0x20000000
D: 0x08000400
E: Undefined, not initialized

Address in memory

0x20000000
0x20000190

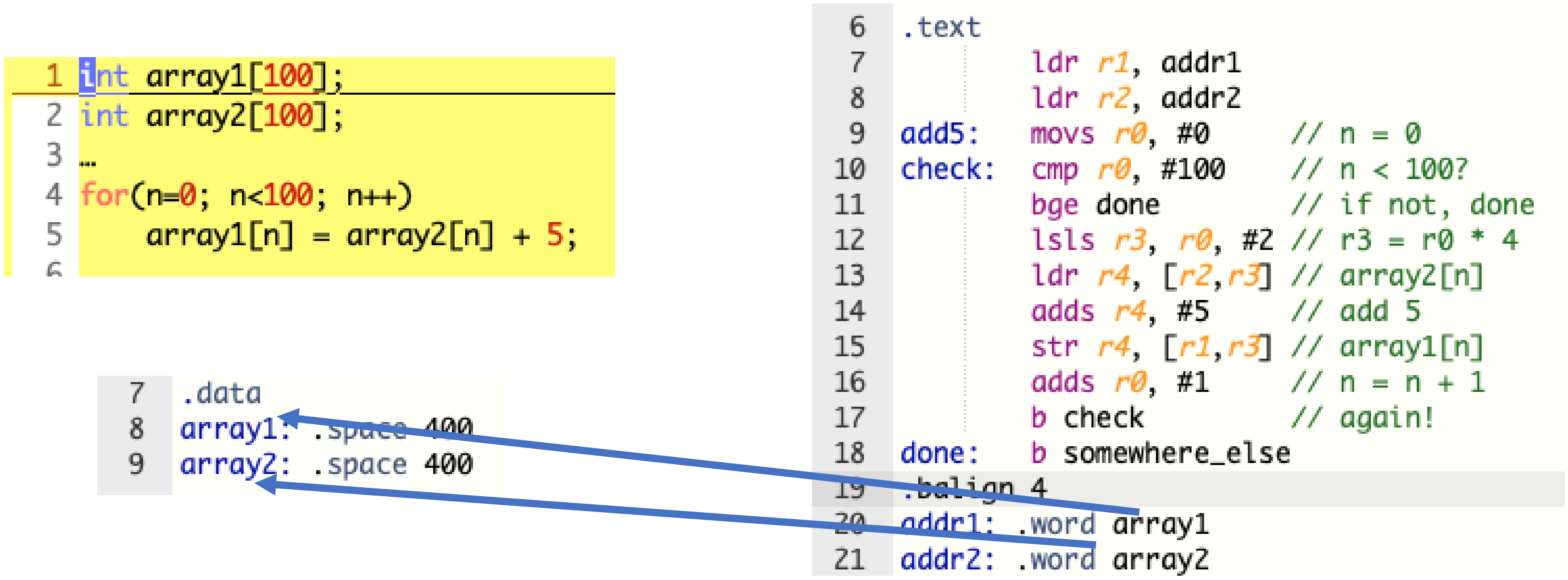
0x08000400
0x08000404



```
.data
array1: .space 400
array2: .space 400


.text
addr1:   .word array1
addr2:   .word array2
```

- addr1/addr2 are locations in text memory where the ***4-byte address*** of array1/array2 are stored
  - They are ***pointers***

Accessing an array from assembly



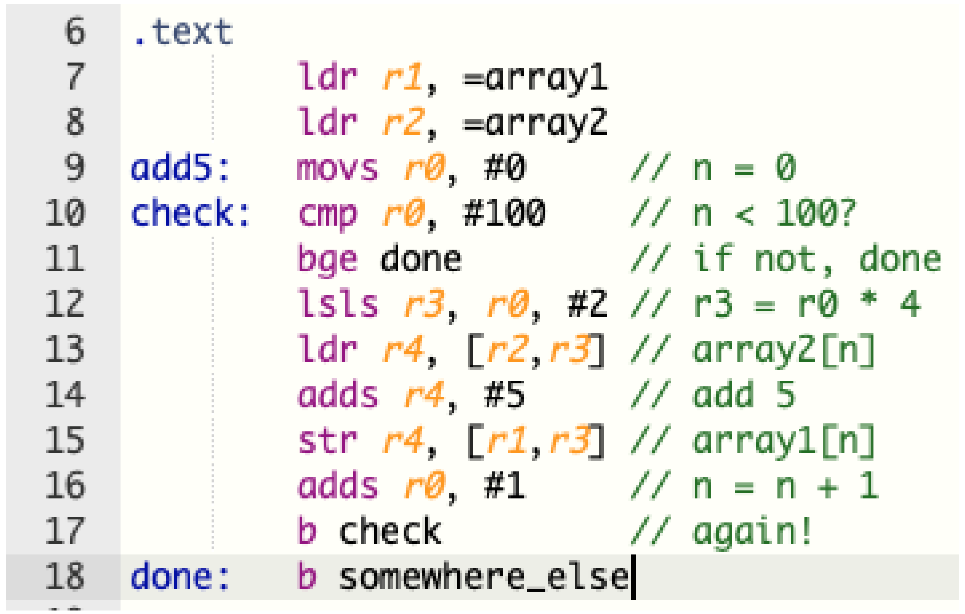
```c
int array1[100];
int array2[100];
…
for(n=0; n<100; n++)
    array1[n] = array2[n] + 5;
```

```
.data
array1: .space 400
array2: .space 400
```

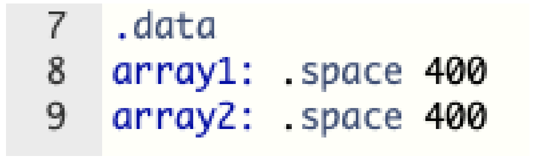
```
.text
        ldr r1, addr1
        ldr r2, addr2
add5:   movs r0, #0      // n = 0
check:  cmp r0, #100     // n < 100?
        bge done         // if not, done
        lsls r3, r0, #2  // r3 = r0 * 4
        ldr r4, [r2,r3]  // array2[n]
        adds r4, #5      // add 5
        str r4, [r1,r3]  // array1[n]
        adds r0, #1      // n = n + 1
        b check          // again!
done:   b somewhere_else
.balign 4
addr1: .word array1
addr2: .word array2
```

So common – assembler has a shortcut

- Just use LDR with a =symbol, and it will automatically build a literal pool at the end of the block of code…



```
.text
        ldr r1, =array1
        ldr r2, =array2
add5:   movs r0, #0        // n = 0
check:  cmp r0, #100       // n < 100?
        bge done           // if not, done
        lsls r3, r0, #2    // r3 = r0 * 4
        ldr r4, [r2,r3]    // array2[n]
        adds r4, #5        // add 5
        str r4, [r1,r3]    // array1[n]
        adds r0, #1        // n = n + 1
        b check            // again!
done:   b somewhere_else
```



```
.data
array1: .space 400
array2: .space 400
```