Cortex-M0 Instruction Set Relative & Indexed Addressing Modes

ECE 362
https://engineering.purdue.edu/ece362/

More reading

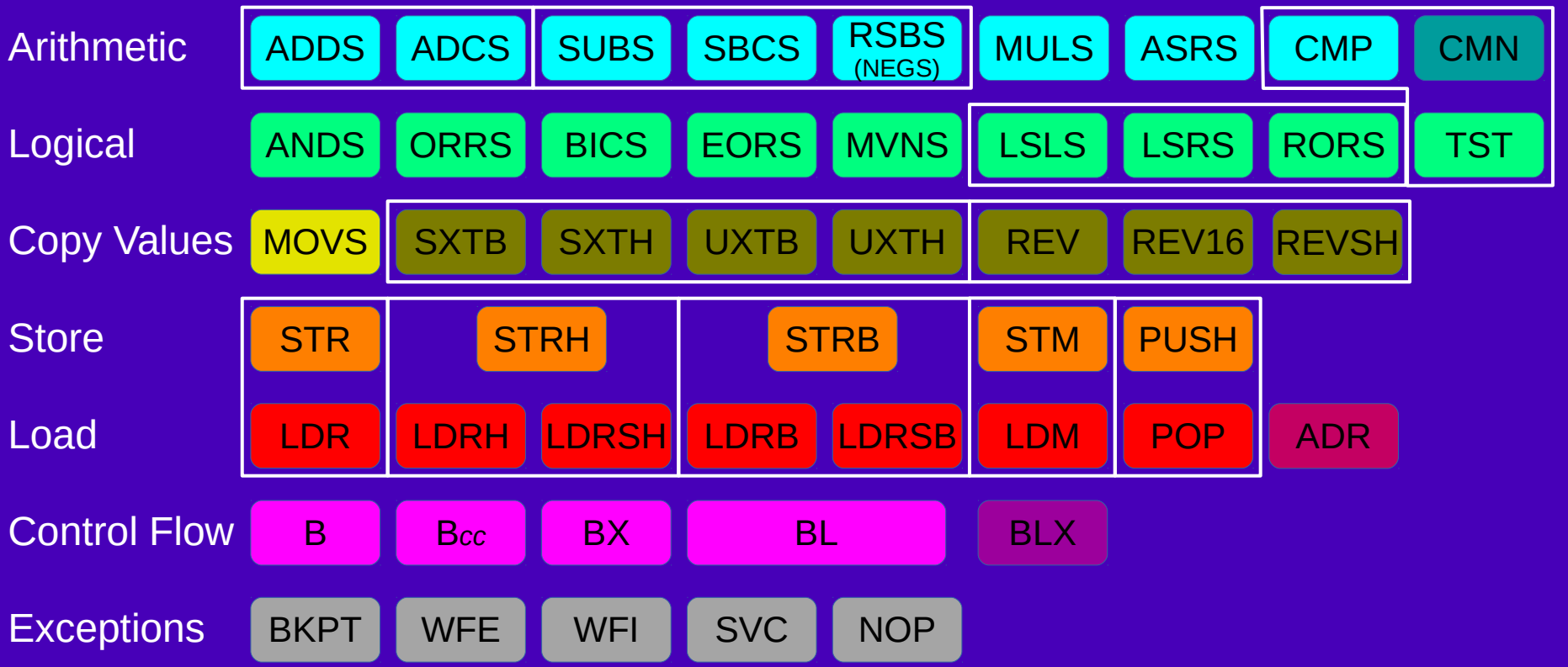
- Textbook, Chapter 6, “Branch and Conditional Execution”, pages 111 – 132.
  - Talking about this in this lecture.
- Textbook, Chapter 5, “Load and Store”, pages 97 – 110.
  - Talking about this in the second half of the lecture.

More reading after that

- 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)

Instruction Synopsis



Relative Addressing Mode

- The CPU will fetch/execute instructions one after another. We need a way to tell the CPU to fetch its next instruction from a different address. That means changing the PC to a new value rather than incrementing it.
- We can't easily *load* a 32-bit value into a register.
- Just as hard to *jump*. There is no jump instruction with absolute addressing mode like we had with the simple computer or x86.
  - Not enough room in a 16-bit instruction to store a 32-bit address.
  - Instead, we have branches that are relative.

The Relative Branch

- Effectively adding a signed immediate value (+/-) to the PC.
- Unconditional and conditional versions.
- NOTE: During the execution of any instruction, the PC always points to the next instruction.
  - A branch to offset 0 would continue with the next instruction.
  - This is true for every architecture I know of. Except ARM Cortex
  - On ARM CPUs, a branch to offset 0 looks like this:

Branch normally used with a label

- We don’t have to care about ARM CPUs being strange.
- We don’t want to have to hand-compute address offsets… ever.
  - Let the assembler do it for us.

```
.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
```

“Example 0” on lecture notes web page

Simpler Branch Example

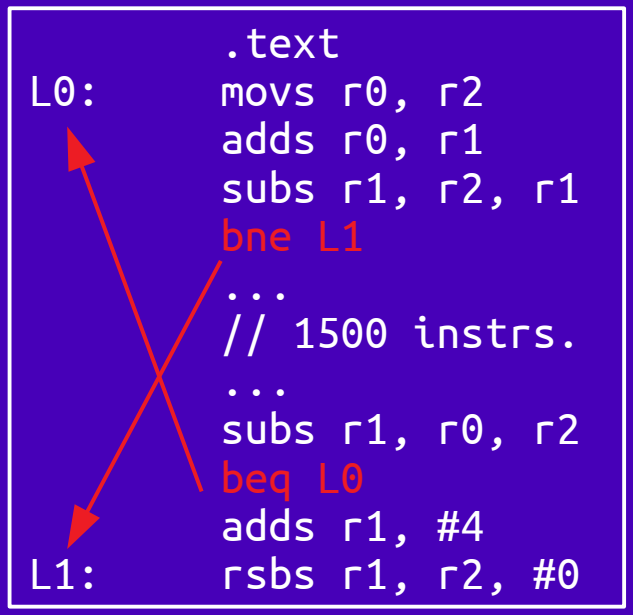
- Now that we can appreciate that the assembler will find branch offsets for us, let's try a simple loop for an example:

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

"Example 1" on lecture notes web page

Longer Branch Range Is Better

- There is a limit to how far forward and backward a branch instruction can redirect execution.
- If a program gets too long, it may not be possible to branch to some part of it.
- We usually want to maximize the *range* of a branch instruction.
  - One way to do that is multiply the offset encoded in the instruction by 2.

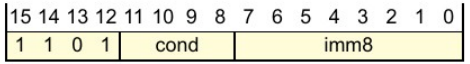


A6.7.10 B

Branch causes a branch to a target address.

Encoding T1 All versions of the Thumb instruction set.

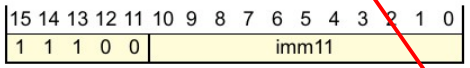
```
B<c>  <label>
```




```
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>
```




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

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);
```

BNE “PC+12”
1101 0001 00000110
0xD106

B “PC-2”
11100 11111111111
0xE7FF

Instructions always start on an even address. So multiply “imm” value by 2 to get byte offset.

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.

How to use conditional branches

- In years past, we used to go into tedious detail about how condition codes are set, and how branches interpret those codes. It's simpler than that:
  - Use an instruction that modifies or tests a value.
    - When you want to compare two integers, that's subtraction.
    - An instruction that sets the flags has an 'S' suffix. (e.g. SUBS)
  - Use a branch that interprets the test in the way you want.
    - Branches conditions are set up to work with subtraction.

Branch Selection Example #1

- Subtract R5 from R2 to give R6. (R6 = R2 - R5)
  - I want to branch to `error` if R2 <= R5.
    - In other words, if R6 is negative or zero.
  - I want to interpret all values as signed, 2's complement integer representation.

```
SUBS    R6, R2, R5
BLE     error
```

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

Branch Selection Example #2

- Subtract R5 from R2 to give R6. (R6 = R2 - R5)
  - I want to branch to `error` if R2 <= R5.
    - In other words, if R6 is negative or zero.
  - I want to interpret all values as unsigned, 2's complement integer representation.

```
SUBS    R6, R2, R5
BLS     error
```

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

Branch Selection Example #3

- I don’t want to subtract anything.
  - I want to branch to `error` if R2 > R5.
    - In other words, if subtraction would be greater than zero.
  - I want to interpret all values as signed, 2’s complement integer representation.

```
CMP     R2, R5
BGT     error
```

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

Wait. Why is it “CMP” rather than “CMPS”?

Branch Selection Example #4

- I don’t want to subtract anything.
  - I want to branch to `error` if R2 >= R5.
    - In other words, if subtraction would be greater or equal to zero.
  - I want to interpret all values as unsigned, 2’s complement integer representation.

```
CMP     R2, R5
BHI     error
BEQ     error
```

1000	HI	Unsigned higher	C == 1 and Z == 0

0000	EQ	Equal	Z == 1

Signed or 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

Signed or Unsigned?

- If we do math on numbers that we want to treat as unsigned, it means we need to change the interpretation of “less than zero”.
  - ADDS, SUBS, and CMP work the same either way.
  - The proper branch must be selected to reflect what we want.

Signed or Unsigned?

- In C, we can explicitly declare variables that are never negative (always positive or zero) like this:

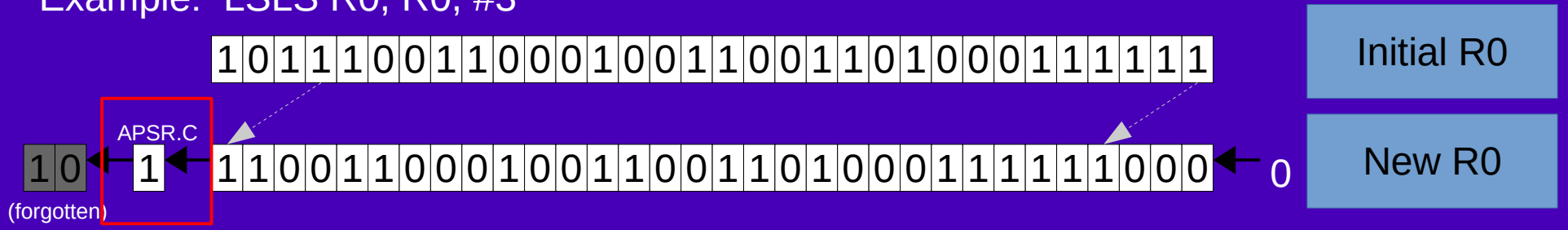
```
unsigned int   big32 = 0xffffffff; // AKA uint32_t
unsigned short big16 = 0xffff;     // AKA uint16_t
unsigned char  big8  = 0xff;       // AKA uint8_t
```

- We'll look at this again when we talk about converting C into assembly language.

Logical Shift Left

- We can shift a 32-bit value in a register “left” by any number of bits.
  - This is the same as the C language operator: <<
  - Bits that are shifted out the left end (MSB) of the register are moved into the APSR Carry flag, so we call it “LSLS”.
  - Zeros are shifted in on the right side (LSB).

Example: LSLS R0, R0, #3



We can use LSL to test bits

- Shift “leftmost” bit into the carry flag.
- How do we test if the carry flag is set?
  - BCS: “Branch if Carry Set”

```
// 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

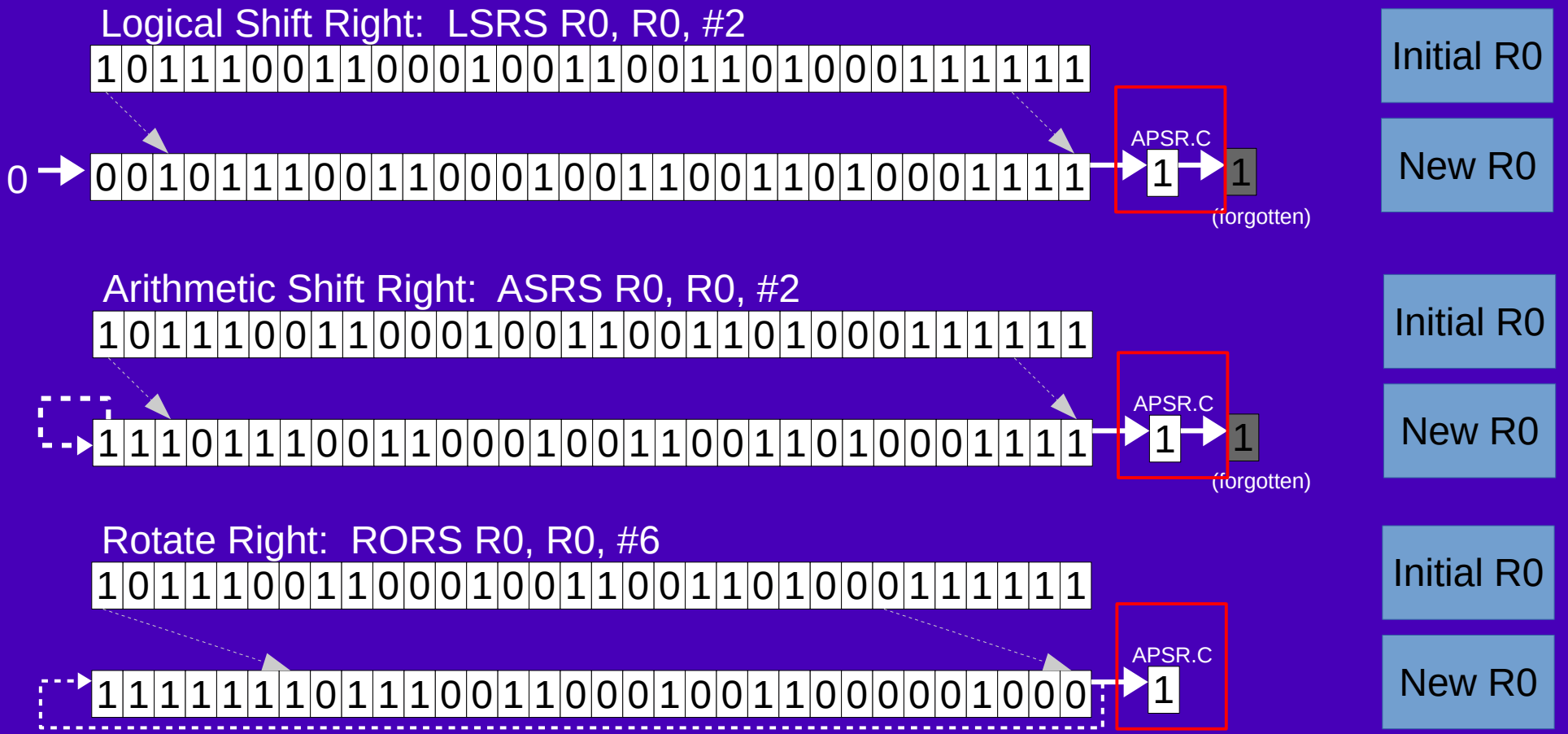
a. HS (unsigned higher or same) is a synonym for CS.
b. LO (unsigned lower) is a synonym for CC.

“Couldn’t we have used this a few slides ago instead of BHI and BEQ?”

Yes! But you wouldn’t have learned as much.

“Example 2” on lecture notes web page

Other Shift & Rotate Instructions



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
0 0 0 0 1 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0 0 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 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
0 0 0 0 1 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0 0 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 0 1 0 0 0 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
0 0 0 0 1 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0 0 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 1 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
0	0	0	0	1	0	0	0	0	0	0	0	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	1	1	

Initial R0 AND R1

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
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

Initial R1

New R0

Loads and Stores Indirect/Indexed Address Mode

- The Cortex-M0 CPU cannot reference memory with arithmetic or logical instructions.
- It can only reference memory with load and store instructions.
  - That's why it's called a load/store architecture.
- It also cannot refer to an absolute address.
  - Instead, it refers to an address *relative* to a register value.
  - A branch is relative to the PC.
  - A load or store is relative to one of registers R0 – R7.

LDR: Load a word of memory into a register

- 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.

STR: Store register into a word of memory

- 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.

Example of Store / Load

- Consider the following program:

```
ldr r0, =0x20000000 // R0 = 0x20000000
str r1, [r0]        // mem[0x20000000] = R1
ldr r1, [r0]        // R1 = mem[0x20000000]
```

Simple Store/Load Example

```
.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
```

- In this example, we initialize R2 to an explicit address (which happens to be the lowest address for SRAM on STM32).

"Example 3" on lecture notes web page

Another Store/Load Example

```
.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
...
storage: .word 0
```

- In this example, we initialize R2 to a label that refers to a location where space is allocated. We don't know where it gets put. The assembler decides for us!

"Example 4" on lecture notes web page

Indirect addressing

- When you see [ ] around something, it means to treat it as an address and do a memory operation.
  - We don't need to only use R0 for an address. We can use R0 – R7.
  - But we've seen in a past lecture that it's hard work to initialize a register to a 32-bit address.
  - Is there a way we can use LDR/STR to refer to memory ***near*** that address?

Indexed Addressing Mode

- LDR/STR take an extra *offset* or *index*
- LDR R1, [R0, #12]
- “Take the value of R0, add 12 to it, use the sum as an address, go to that address, read four contiguous bytes, and put it in R1.”
- The offset can be any multiple of 4 from 0 – 124.
- Similar specification for STR.
- Caveat: R0+offset must be evenly divisible by 4.

Indexing with a register

- I know what you're going to ask:

  "Can I index with a register instead of an immediate value?"
- LDR 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.

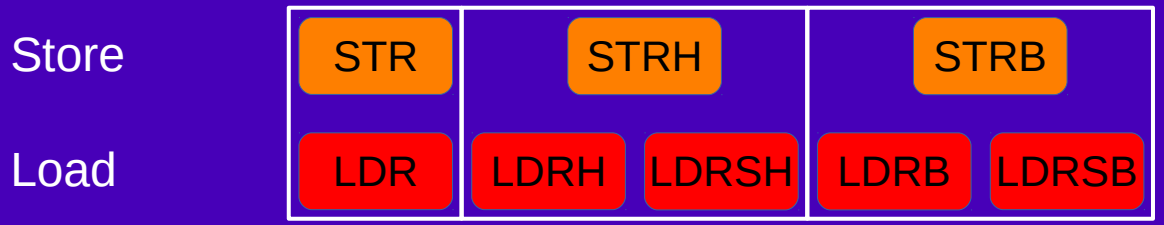
Also byte/halfword load/store

- 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 are there two kinds of loads?

- Why have LDRH / LDRSH and LDRB / LDRSB when we only have STRH and STRB?



- Registers are always treated as 32 bit values.
  - There is no instruction to “add only the lower 16 bits of registers Rx and Ry”
- When loading an 8-bit value from memory, we need to know if it is supposed to be treated as signed or unsigned. (i.e., should bit 7 indicate it is negative)
  - If we load `0xff` from memory using LDRB, it will set the destination register to `0x000000ff`
  - If we load `0xff` from memory using LDRSB, it will set the destination register to `0xffffffff`.

Store Byte/Load Byte Example

```
.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 is stored in memory.
- It is loaded from memory as a *signed* byte and then incremented.

"Example 5" on lecture notes web page

Reading and writing arrays

- How can we implement the following C code:

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

For the next few minutes, assume that:
n is just register R0,
R1 is magically the address for array1,
R2 is magically the address for array2.

```
.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
```

How to load literal values

- We can jump (update the PC) to a relative offset from the current PC value.
- We can load a value from a relative offset from the current PC value.
- LDR R1, [PC, #12]

  "Take the current PC value, round it up to a multiple of 4, add 12 to it, use it as an address, read 4 contiguous bytes from memory, and put the result in R1."
- Can we store to an address relative to the PC?
  - No. The PC normally points to read-only memory where the instructions are.
- Bonus: PC (rounded up) +12 is automatically evenly divisible by 4.
  - But the data you refer to might not be. More on that in a bit.
- We almost always use the preferred form LDR R1, label
- Caveat: The item must be no further than 1020 bytes beyond the PC in the **text** segment.

Example of literal load

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


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

Maximum offset: 1020 bytes.
Word must be in the text segment.
Instructions are 2 bytes long.
Word must be 4-byte aligned.
How do we guarantee alignment?

Label is an address

- What value is put into the memory words allocated for addr1 and addr2?

```
            .data
0x20000000 array1:  .space 400
0x20000190 array2:  .space 400


            .text
0x08000400 addr1:   .word array1→0x20000000
0x08000404 addr2:   .word array2→0x20000190
```

array1 is the location in memory of a 400-byte reservation that is uninitialized.

Maybe array1 is in the data segment.

addr1 is the location in memory of a 4-byte reservation initialized to be the address of array1.

Maybe addr1 is in the text segment.

Values like this in the text segment are called a “literal pool”

Reading and writing arrays

- Consider this C code again:

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

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

```
        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
```

If that still seems like too much work...

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

```
        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
```