Cortex-M0 Instruction Set
Relative & Indexed
Addressing Modes

ECE 362
https://engineering.purdue.edu/ee362/

© Rick
More reading

  - Talking about this subject today.
  - Talking about this subject Friday.
More reading after that

  - Talk about this next week.

  - Talk about this late next week.
Relative Addressing Mode

- We can’t easily \textit{load} a 32-bit value into a register.
- Just as hard to \textit{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.
  - On ARM CPUs, a branch to offset 0 looks like this:

```
instr instr instr B (0) instr instr instr instr instr
```

Except ARM
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
L0:     movs r0, r2          0x0010
       adds r0, r1          0x4408
       subs r1, r2, r1     0x1a51
       bne L0              0xd1fb
       subs r1, r0, r2     0x1a81
       beq L1              0xd000
       adds r1, #4         0x3104
L1:     rsbs r1, r2, #0     0x4251
```
Instructions always start on an even address. So “imm” values are half of the actual byte offset.
How to use conditional branches

- 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 comparing 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.
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.

<table>
<thead>
<tr>
<th>SUBS</th>
<th>R6, R2, R5</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLE</td>
<td>error</td>
</tr>
</tbody>
</table>
Example #2

- Subtract R5 from R2 to give R6. \( (R6 = R2 - R5) \)
  - I want to branch to **error** if \( R2 \leq R5 \).
    - 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
```
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 negative or zero.
- I want to interpret all values as signed, 2’s complement integer representation.

```
CMP R2, R5
BLE error
```

Wait. Why is it “CMP” rather than “CMPS”??
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 “LSL$S$”.
  - Zeros are shifted in on the right side (LSB).

Example: `LSLS R0, R0, #3`

```
Initial R0: 1011100111000100110011010010001111111
APSR.C: 10

New R0: 11001100011001100110100011111111111000
```
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”

```c
// 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:
```
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.
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 addresss 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.
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
  - LDRB: load a byte [offset is any byte from 0 - 31]
  - LDRH: load a halfword (2 bytes) [offset any mult of 2 from 0 - 62]
  - LDRSB: load a signed byte (sign-extend bit 7) [offset same as LDRB]
  - LDRSH: load a signed halfword (sign-extend bit 15) [offset like LDRH]
  - STRB: store a byte [offset is any byte from 0 - 31]
  - STRH: store a halfword [offset is any multiple of 2 from 0 - 62]

- Caveat: Halfword load/stores must use an address that is evenly divided by 2.
Reading and writing arrays

• How can we implement the following C code:

```c
int array1[100];
int array2[100];
...
for(n=0; n<100; n++)
    array1[n] = array2[n] + 1;
```

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.

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

.text:
    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, #1     // add 1
    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, and put the result in R1.”
• Can we store to an address relative to the PC?
  – No. That’s usually read-only memory.
• Bonus: PC+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: Data must be no further than 1020 bytes beyond the PC.
Example of literal load

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

.align 4
bigvalue:    .word 0xfedcbe98
value2:      .word 0xdecf123
```

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?

```assembly
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 initialized. Maybe it is in the data segment.

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

```c
int array1[100];
int array2[100];
...
for(n=0; n<100; n++)
    array1[n] = array2[n] + 1;
```

```assembly
ldr r1, addr1
ldr r2, addr2
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, #1     // add 1
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
.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...

```assembly
ldr r1, =array1
ldr r2, =array2
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, #1     // add 1
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