# DIGITAL SYSTEMS TT2019

# SHEET 6

GABRIEL MOISE

## Question 1

In this question I will rely on "Figure 17: Decoding table for all instructions" from the hand out. Furthermore, let's recall that

- Rd = <2:0>, Rn = <5:3>, Rm = <8:6>, Rt = <10:8>, RHn = <6:3>, RHd = <7> ++ <2:0>
- Imm8 = <7:0>

a) Case 1 : cRegSelC = Rd

This will not affect the instructions which already have the decoding with cRegSelC = Rd or those which don't care about the value of the register C because they do not use it, those that have "-" in the cRegSelC value in the table. So, the cRec will always be the one decoded from bits <2:0> in this case and the instructions which will still work **correctly** will be:

lsls i5, lsrs i5, asrs 15, adds/subs i3, cmp i8, ands r, eors r, lsls r, lsrs r, asrs r, adcs r, sbcs r, rors r, tst r, negs r, cmp r, cmn r, orrs r, muls r, bics r, mvns r, cmp hi, str r, ldr r, str i5, ldr i5 (underlined those with "-")

All the others will behave incorrectly because they will use the wrong bits for the third register (cRegC) and it has an impact on the result.

Case 2 : cRegSelC = Rt

Analogously to the case above, the instructions that will still behave **correctly** will be:

movs i8, cmp i8, adds i8, subs i8, tst r, cmp r, cmn r, cmp hi, ldr pc, str sp, ldr sp, add pc, add sp

b) Case 1 : cRand2 = RegB – chooses rb as the output of rand2sel

When this happens, all the instructions that have cRand2 = RegB already behave correctly, the other instructions that use immediate fields instead can't work correctly, since they need to get the bits from the instruction, and not the value from the register b, which will always be chosen if cRand2 will be stuck on RegB in the rand2sel multiplexer. So, the instructions which behave **incorrectly** (there are a few compared to the others) are:

adds/subs i3, movs i8, cmp i8, ads i8, subs i8, ldr pc, str i5, ldr i5, str sp, ldr sp, add pc, add sp, sub sp, b <c>, b, bl1, bl2

Case 2 : cRand2 = Imm8

This way, in the rand2sel multiplexer we get to choose the first 8 bits from the current instruction instead of the value from rb, so it's obvious that the instructions that need that will work **correctly**, which are:

movs i8, cmp i8, adds i8, subs i8, ldr pc, str sp, ldr sp, add pc, add sp, b<c>

c) Case 1 : cMemRd = F

This would mean that the instructions that did not need any reading from the memory will stay the same and will behave correctly, whereas the others will not be able to do what they are supposed to do because they lack the information they need from the memory stack. Those which will work **incorrectly** are:

ldr pc, ldr r, ldr i5, ldr sp (they are all load instructions, so they need to read from the memory to be able to put the values they need in the specified register)

Case 2 : cMemRd = T

Again, obviously the instructions above (the load instructions) will work as intended, since now they got the information they needed, but the other do not need any reading from the memory as part of their task. The problems are that this sometimes causes a cache miss or an exception for an invalid address, as we can discover in the lecture, but the cMemRd is also responsible for what gets put in the result. If it's always true, the "result" multiplexer will always choose the result from "memout" and this will not always be the result we want, but what was loaded from memory in the memory location specified from alout. So, the only instruction that will behave **correctly** are:

ldr pc, ldr r, ldr i5, ldr sp

d) Case 1 : cWReg = N

Every instruction that needs to write in a register can't do that anymore, so registers can't be modified anymore (except for pc and maybe lr), so only the following instructions will be the only ones to behave **correctly** are:

cmp i8, tst r, cmp r, cmn r, cmp hi, str r, str i5, str sp, b <c>

Case 2 : cWReg = Y

If we always write in the registers the result from the instructions above, we will modify them without wanting that and we will most probably get wrong results as we go on with the program. So, the instructions that will behave **incorrectly** will be:

cmp i8, tst r, cmp r, cmn r, cmp hi, str r, str i5, str sp, b <c>

e) Case 1 : cWFlags = F

In this case, we never update the flags no matter what happens, so all the instructions that should generally modify them will behave incorrectly, and the other ones, although they do what they are supposed to do (and not modify the flags), most of them rely on their correct values, so, although they will do what they should, given the input, they will most probably do the wrong things determined from previous instructions. So, the instructions that will behave **correctly** are:

add hi, mov hi, bx/blx r, ldr pc, str r, ldr r, str i5, ldr i5, str sp, ldr sp, add pc, add/sub sp, b<c>, b, bl1, bl2

Case 2 : cWFlags = T

In this case, the flags will be updated every time and therefore the instructions that shouldn't modify the values of the NZVC flags will do the wrong thing, and as we saw in the lecture notes, this might affect programs where we compare something, and then use 2 conditional-branches afterwards. So, this functions will behave **incorrectly** in this situation:

add hi, mov hi, bx/blx r, ldr pc, str r, ldr r, str i5, ldr i5, str sp, ldr sp, add pc, add/sub sp, b<c>, b, bl1, bl2

## Question 2

The address to where pc should be, will be formed by adding the 1(the sing bit) + 10 (offset of bl1) + 11 (offset of bl2) = 22 bits, extended from the sign bit to the value of pc. The process is:

- we put the value of [pc + the 10-bit offset from bl1, with sign extension of S, which is the 11th bit, so we get pc + S from <31:22> ++ Offset10 from <21:12> ++ 0 from <11:0> (because we can see that in the bl1 instruction the shifter will perform a left shift with 12 positions)] into the lr and pc gets to the next instruction, which is bl2
- we add to the value of lr, which is the one we calculated above, the Offset11 from bl2, without being sign extended(if we sign-extended this, too, in the case where the last bit was going to be a 1, we would not combine properly the two halves of the offset, since the sign bit of the offset is already in the offset of bl1, so we would get a wrong result in this case), shifted by 1 position, so the new value of lr will be [pc+ S from <31:22> ++ Offset10 from <21:12> ++ Offset11 from <11:1> and the first bit is 0 (in the bl2 instruction we shift left the offset by 1 position, so that the value added to pc is even, so that pc doesn't get an odd value] and it is put in pc, and pc goes to the place where the branching is made

## Question 3

a) Choosing the maximum of two values from registers r0 and r1 and putting the result in r2 (so that we don't lose any of them) will be implemented with conditional moves as:

```
        movs    r2, r0
        cmp     r0, r1
        movlt   r2, r1
        …
```

which always requires 3 clock cycles, whereas the method that doesn't involve conditional moves is:

```
        cmp     r0, r1
        bge     first
        movs    r2, r1
        b       second
first:  movs    r2, r0
second: …
```

which, if r0 >= r1 it will need 3 clock cycles, but if r0 < r1, it will need 4 clock cycles, so, on average it needs 3,5 cycles, so our first implementation is slightly better and the introduction of conditional moves is justified.

b) An encoding for the conditional move in Thumb move could be

- <15:12> - bits for decoding the conditional move instruction : 1100 (we can use 24 and 25 as indicated)
- <11:10> - bits to determine the shift amount (Sh0/Sh1/Sh2/Sh12)
- <9:6> - bits to determine what condition we have so that we can choose the corresponding flags
- <5:3> - Rn = register to be moved if the condition is satisfied
- <2:0> - Rd = register destination

The entry for the decoding table would be:

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| mov<c> | - | Rn | Rd | RegB | Lsl | Sh0 | Mov | F | F | F | C | N |

c) We can implement almost every instruction as conditional, but I think that the str instruction cannot be implemented with the current datapath because we can't restrict the overwriting in the memory stack with the flag-conditions.

## Question 4

a) The addressing mode

```
ldr r0, [r2, r3, LSL #2]
```

is particularly useful in programs that contain a lot of array indexing because if we have the array location in memory, which will be in r2, and if we want to access the ith position in the array, to obtain the value in it, and then load that value from memory in r0, we would normally do that:

lsls r1, r3, #2 @ each entry of the array has allocated 4 bytes of memory on the stack, so we shift by 2 positions

ldr r0, [r2, r1] @ we get the position in the memory stack that we wanted because r2 is the base address of the array

but with the first addressing mode, this is done in only one clock cycle and therefore it is twice faster.

b) The decoding rules for versions of the ldr and str instructions that implement this addressing mode would be:

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ldr | Rn | Rm | Rd | RegB | Lsl | Sh2 | Add | T | F | F | Y | N |
| str | Rn | Rm | Rd | RegB | Lsl | Sh2 | Add | F | T | F | N | N |

with the syntaxes:

```
ldr Rd, [Rn, Rm, LSL #2]

str Rd, [Rn, Rm, LSL #2]
```

and the encodings:

An encoding for the str/ldr instructions would be:

- <15:11> - bits for the encoding of the instruction: 01110/01111 (op code 14 & 15 as suggested)
- <10:9> - bits that can be used for the shift part, which can be either Sh0/Sh1/Sh2/Sh12\
- <8:6> - Rm = “index” register (the array case)
- <5:3> - Rn = “base address” register
- <2:0> - Rd = destination register