Assembly III

ECE 3710

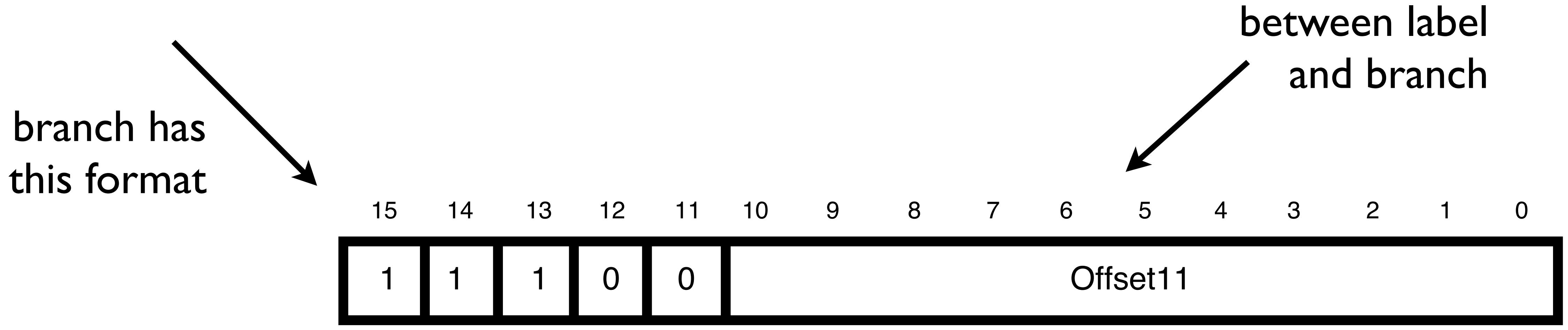
Cross country skiing is great if you live in a small country.

- Steven Wright

Thumb2 instructions: 16-bit

```
00000000 6808          label1 LDR R0,[R1]
00000002 F101 0104            ADD R1, #4
00000006 E7FB                 B  label1
```

Q: how to get this from this



Thumb2 instructions: 16-bit

```
00000000 6808          label1 LDR R0,[R1]
00000002 F101 0104            ADD R1, #4
00000006 E7FB                 B  label1
```

offset = 0x0 - 0x6 - 4 ← PC will be one/two instructions ahead b/c of pipeline

=-10

=0b111111110110 → 111111110110

(two's complement)

12-bits

00000001001

+1

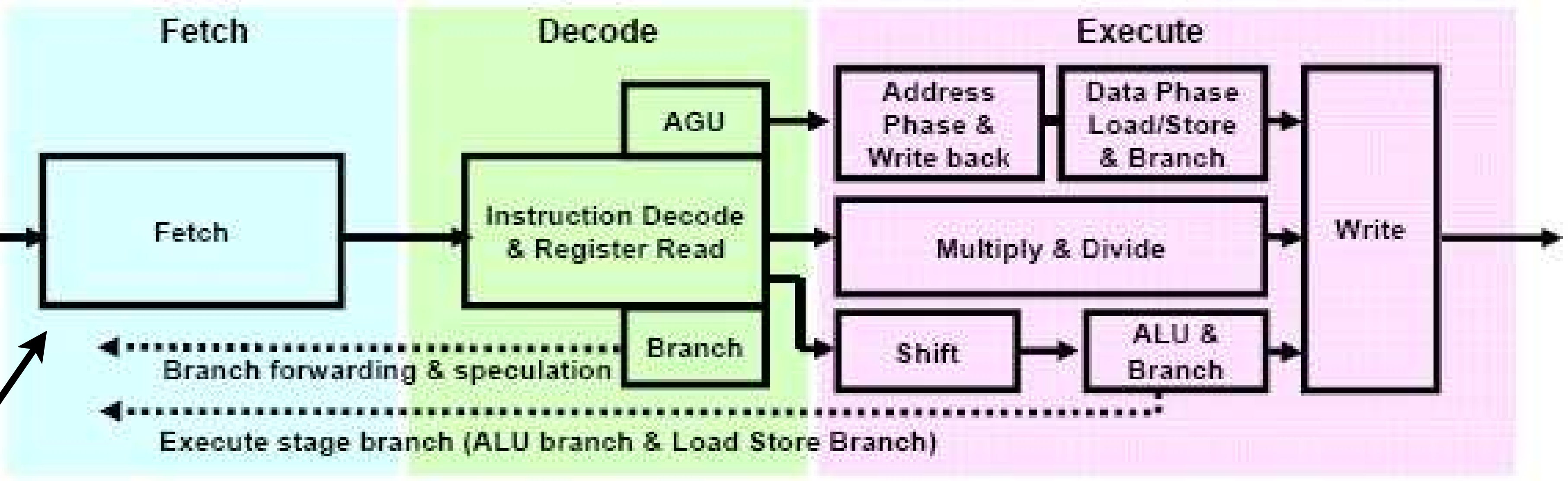
0000 0000 1010

= -10

Thumb2 instructions: 16-bit

Cortex-M3 pipeline:

(three deep)


Fetch
Decode
Execute
Fetch
AGU
Instruction Decode & Register Read
Branch
Address Phase & Write back
Data Phase Load/Store & Branch
Multiply & Divide
Shift
ALU & Branch
Write
Branch forwarding & speculation
Execute stage branch (ALU branch & Load Store Branch)

prefetch unit allows buffers three words (execute 32-bit instructions in single cycle)

Thumb2, too

inst. to be fetched

ARM	Thumb		
PC	PC	FETCH	Instruction fetched from memory
PC - 4	PC-2	DECODE	Decoding of registers used in instruction
PC - 8	PC - 4	EXECUTE	Register(s) read from Register Bank Shift and ALU operation Write register(s) back to Register Bank

offset for two 32-bit ARM instructions

inst. being executed

Thumb2 instructions: 16-bit

(encoded) offset: 11-bits

16-bits

$$0xE7FB = 0b\underline{11100}\overline{11111111011}$$

COND: always
(from ARM)

right shift one
(b/c of half-word align, last bit always zero)

(orig) offset = 0b111111110110
(12-bits; two's complement)

Thumb2 instructions: 16-bit

(coded) offset: 11-bits

$0xE7FB = 0b11100\overline{11111111011}$

decode: left shift one

offset = 0b111111110110

(12-bits; two's complement)

PC for B label1 execution

-10

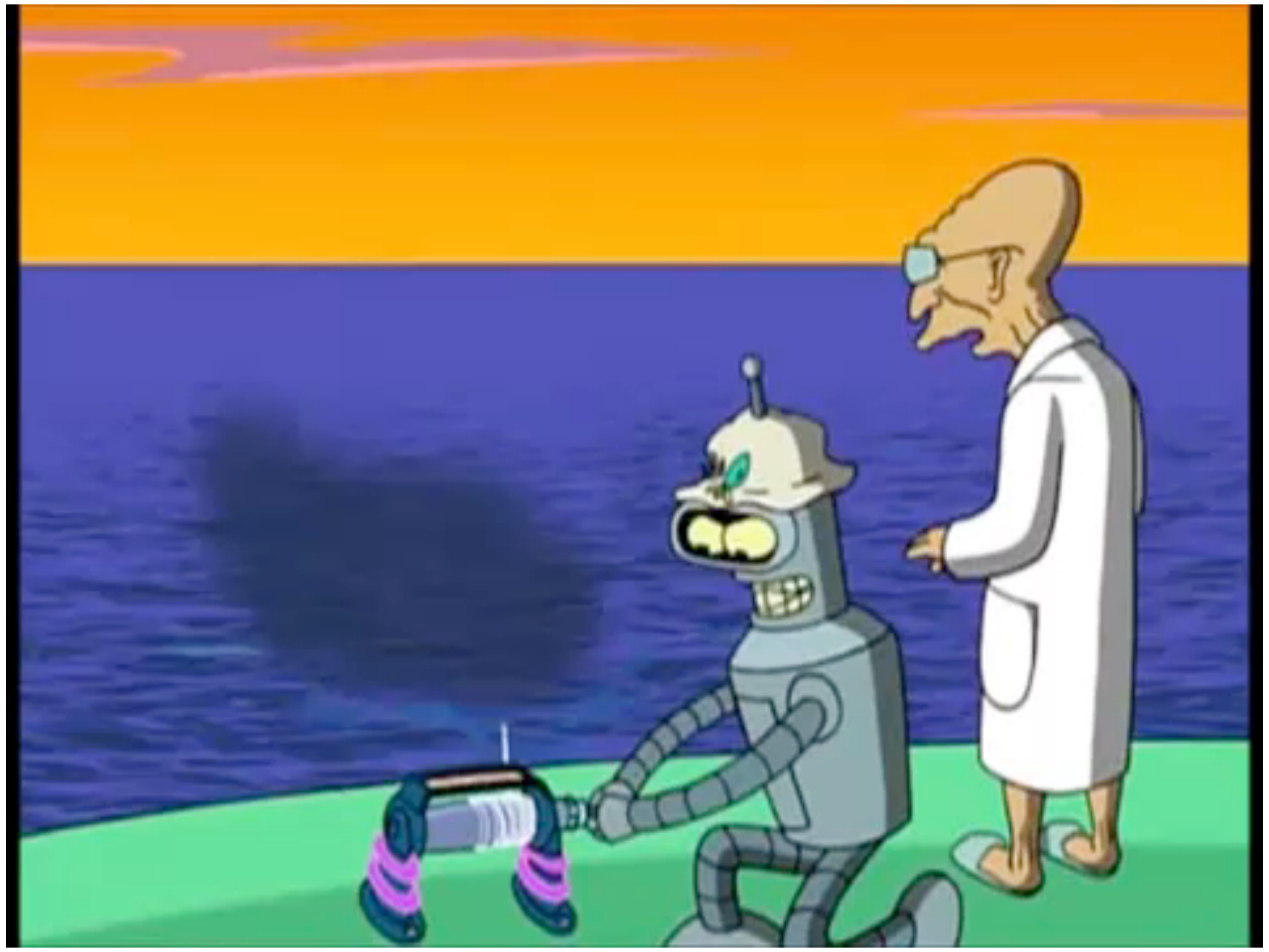
branch to address: 0xA+0b111111110110=0x0

```
00000000 6808        label1 LDR R0,[R
00000002 F101 0104          ADD R1, #
00000006 E7FB               B  label1
```

Thumb2 instructions: 32-bit

don't ask:

```
00000002 F1010104 ADD R1, #4
```

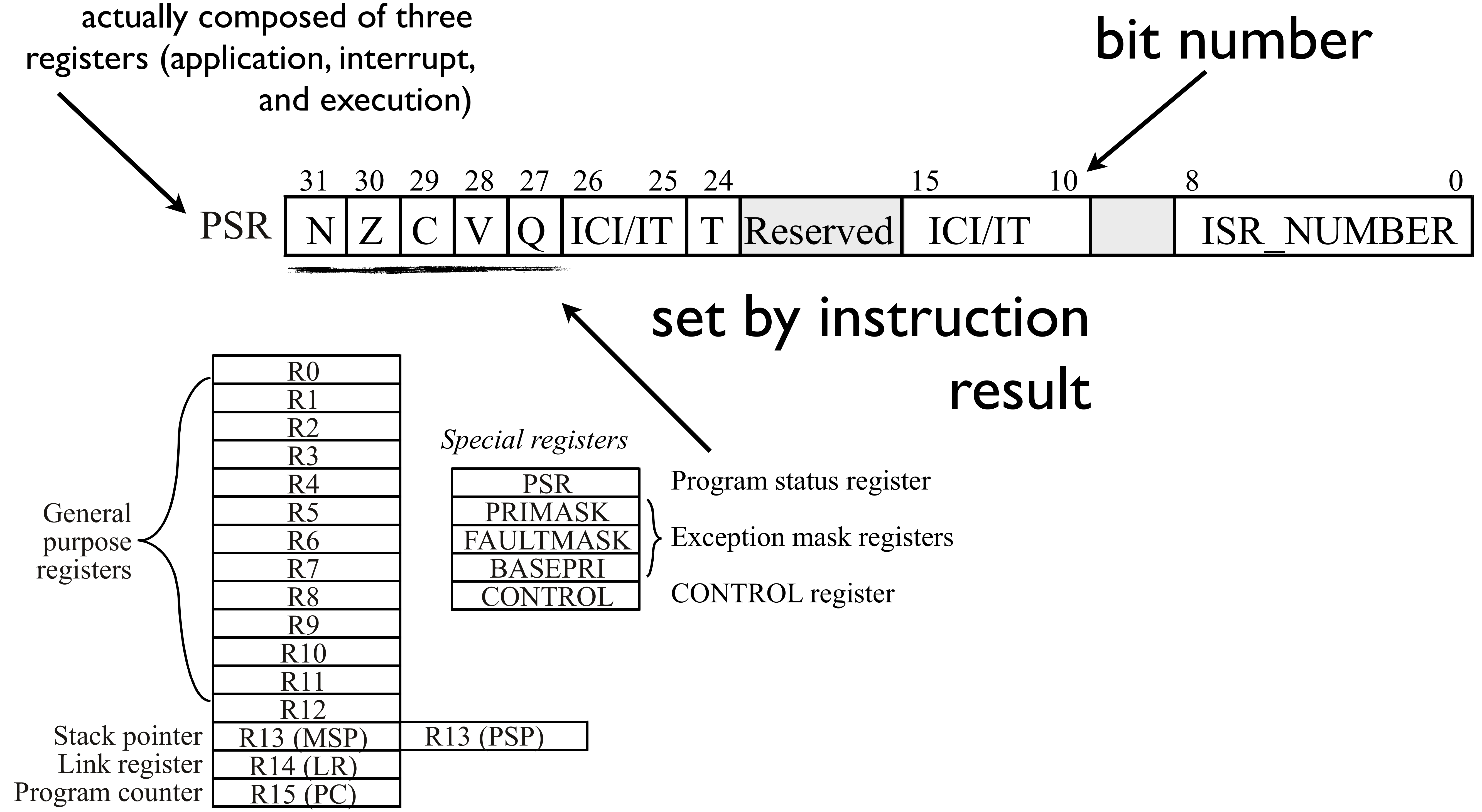


32-bit Thumb is what we have assemblers for...

(if only it were plain ARM...)

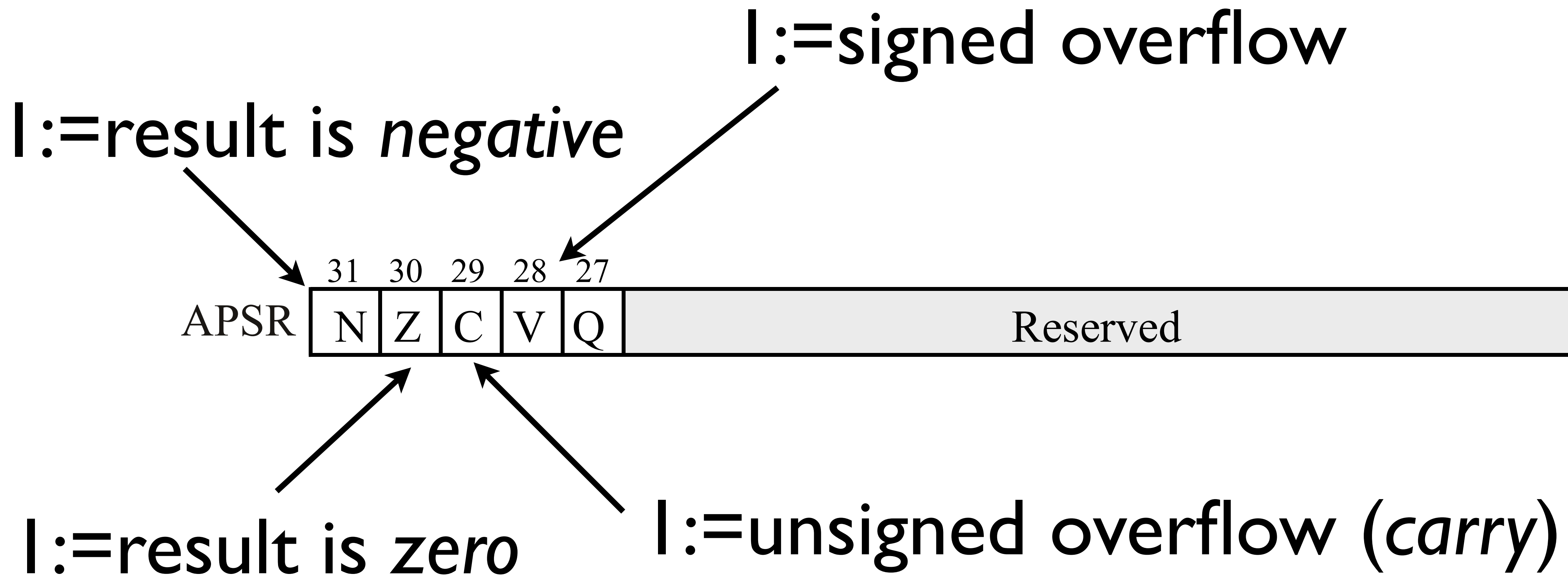
special registers: program status register

(how to achieve conditional branching)



APSR

last operation:



‘condition codes’

making use of APSR:

1. add `{S}` to instruction (later)
2. add `{COND}` to instruction (later)
3. `cmp` and `B{COND}`

a. if X==Y, `Z=1`
b. if X != Y, `Z=0`

i.e. `Z` set to this

conditional branching

```
if R0 == R1
  goto label1;
else
  goto label2;
```

$\longrightarrow$

```
mov R0,#0xA ;init R0
mov R1,#0xB ;init R1
cmp R0,R1   ;R0 ?= R1
beq label1  ;if R0==R1
b label2    ;if R0!=R1
...
label1 ...
       b END
label2 ...
```

syntax:

```
     B{COND} label
CMP Rn, <Operand2>
```

conditionals

`{COND}`

for condition to be met flag must be

e.g.:

`BNE`
`BGE`

Symbol	Condition	Flag
EQ	Equal	Z set
NE	Not equal	Z clear
CS/HS	Carry set/unsigned higher or same	C set
CC/LO	Carry clear/unsigned lower	C clear
MI	Minus/negative	N set
PL	Plus/positive or zero	N clear
VS	Overflow	V set
VC	No overflow	V clear
HI	Unsigned higher	C set and Z clear
LS	Unsigned lower or same	C clear or Z set
GE	Signed greater than or equal	N set or V set, or N clear and V clear (N == V)

Symbol	Condition	Flag
LT	Signed less than	N set and V clear, or N clear and V set (N != V)
GT	Signed greater than	Z clear, and either N set and V set, or N clear and V clear (Z == 0, N == V)
LE	Signed less than or equal	Z set, or N set and V clear, or N clear and V set (Z == 1 or N != V)
AL	Always (unconditional)	—

note: `cmp` sets `NZCV`

conditional branching $\longrightarrow$ loops

```
for(i=0;i<=100;i++)
   j++;
```

$\searrow$

```
mov R0,#0              ;init i
loop cmp R0,#100 ;R0 ?= 100
     beq exit          ;R0 == 100
     add R0,#1         ;R0 != 100 so i++
     add R1,#1         ;j++
     b loop            ;R0 != 100 so loop
...
exit ...
```

note:
`R0 := i` and `R1 := j`

it’s loopy, loopy, man

$\downarrow$

Q: c2asm

```
int h = 12;
int k = 34;
for(int i=0;i++;i<h)
   for(int j=0;j++;j<i)
      k+=56;
```

BLing is like a bungee branch (JMP)...you come back

```
00000000 F04F 0000 Start mov R0,#0
00000004 F000 F804       BL addone
00000008 F100 0009       add R0,#9
0000000C 4902            ldr R1,=theend
0000000E 4708            BX R1
00000010 F100 0001 addone add R0,#1
00000014 4770              BX LR
00000016 E7FE      theend b theend
```

syntax:

1. `BL <label> ; branch to <label>` (+/-16 MB)

long, long branch $\longrightarrow$ 2. `BX Rm ;branch to addr in Rm` (+/-4 GB)

```
00000004 F000 F804          BL addone
00000008 F100 0009          add R0,#9
...
00000010 F100 0001 addone  add R0,#1
00000014 4770               BX LR
```

Q: return to what?

A:

upon BL <label>:

LR := addr. of next instruction

e.g. `LR=0x8=0b1000|1=0x9`

actually: `LR=<addr> | 1`

(this means we stay in Thumb mode; |0 revert to ARM inst)

what about multiple BLs?

```
00000000 F04F 0000 Start mov R0,#0x0
00000004 F000 F803       BL add1      ← LR=0x9
00000008 F04F 000F       mov R0,#0xF
0000000C E7FE      loop  b Loop

0000000E F100 0001 add1  add R0,#1
00000012 F000 F801       BL add2      ← LR=0x17
00000016 4770            BX LR

00000018 F100 0002 add2  add R0,#2
0000001C 4770            BX LR
```

↑ no good, man

(we be stuck)

Wednesday, September 11, 13

what about multiple `BL`s?

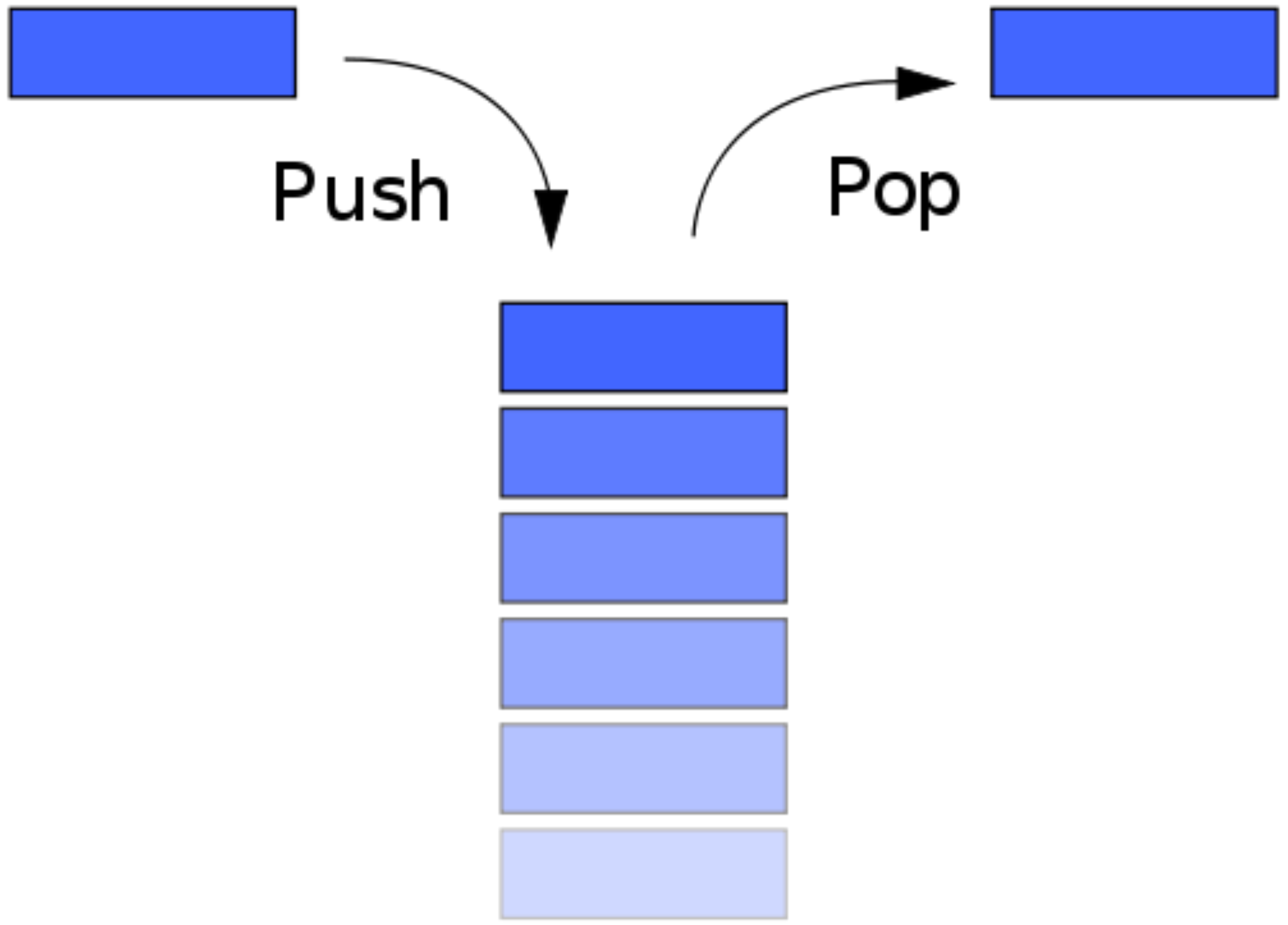
solution:

every time we `BL` we store `LR` on the *stack*

natural response, on learning about the stack:

why the stack?

structure: LIFO (last in first out)
operations: push & pop



for: temporary data (you put it on, you take it off)

using the stack:

0. SP points to bottom of stack

1. push:

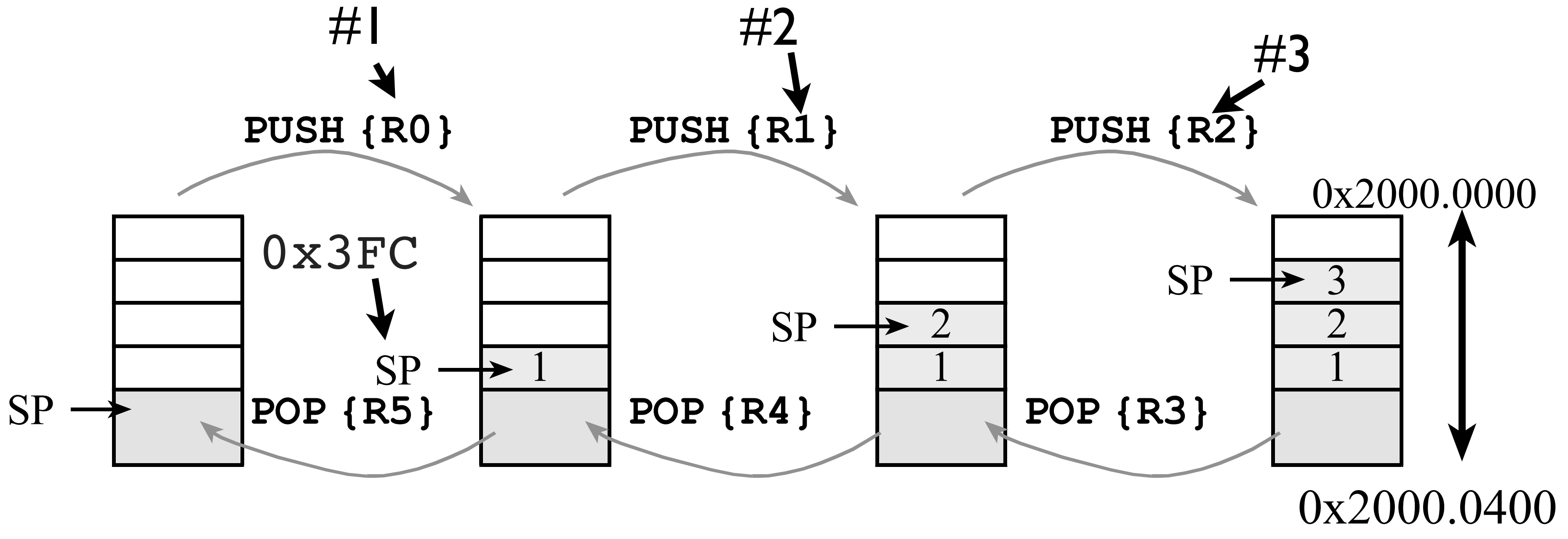
a. SP=SP-0x4

b. store word at SP

2. pop:

a. get word at SP

b. SP=SP+0x4



note: the stack grows down