Assembly II

ECE 3710

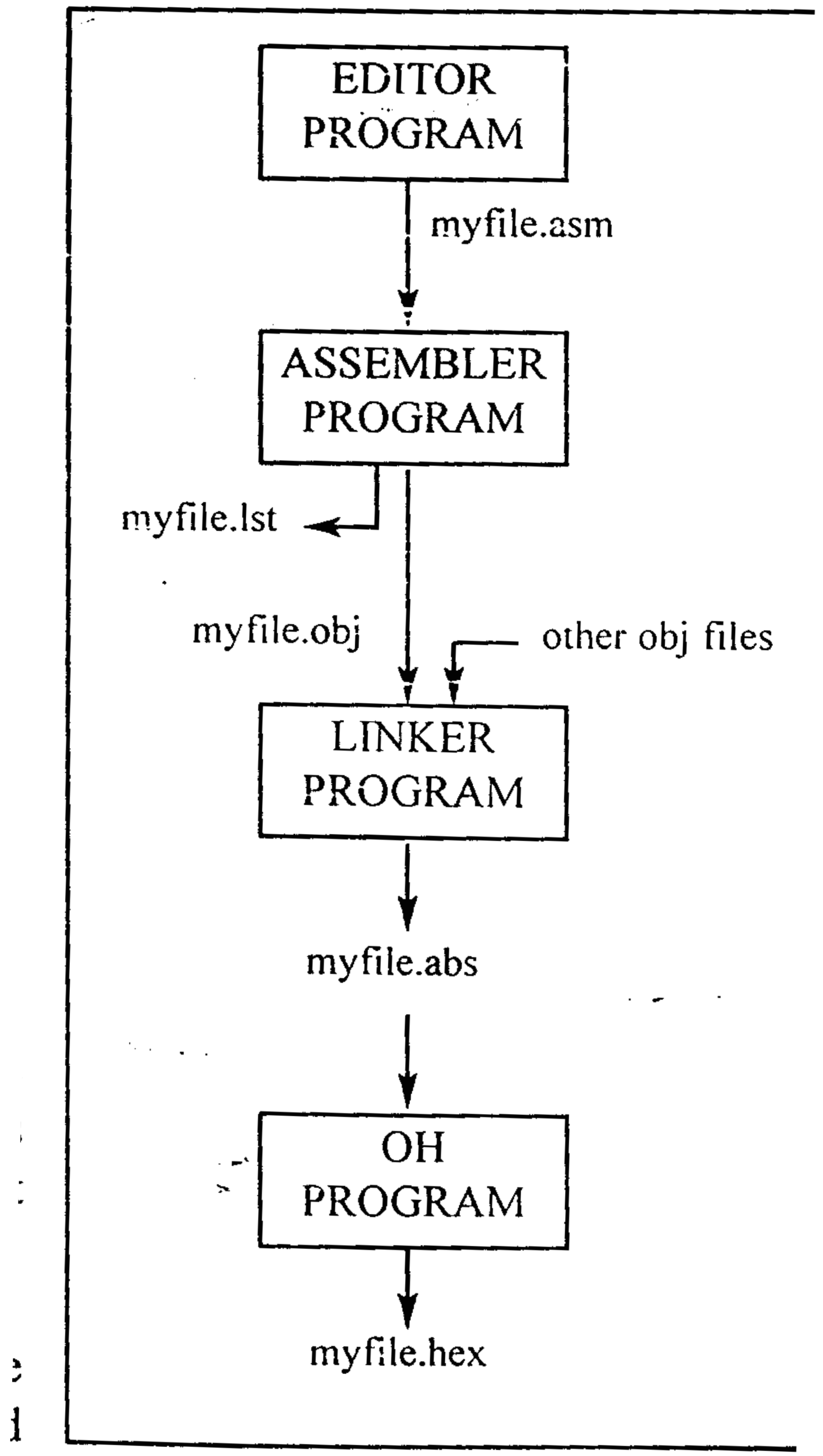
Never ascribe to malice,
that which can be explained
by incompetence.

- Napoleon Bonaparte

lab: ambivalent

do you speak binary?

```
mov R5,#0x25
mov R6,#0x34
add R7,R5,R6
```



from *Keil* assembler

```
00000000: F04F0525 mov R5,#0x25
00000004: F04F0634 mov R6,#0x34
00000008: EB050706 add R7,R5,R6
```

.lst

where   op code.   asm

Q: what can we glean from op code?
registers? immediates?

from *Keil* assembler

```
00000000: F04F0525 mov R5,#0x25
00000004: F04F0634 mov R6,#0x34
00000008: EB050706 add R7,R5,R6
```

.lst

where op code. asm

Q: what can we glean from op code?
registers? immediates?

A:

`0xF04F` indicates operation
next is register number
final is immediate value

see *THUMB Instruction Set*

directives: make the assembler do it (computer < human)

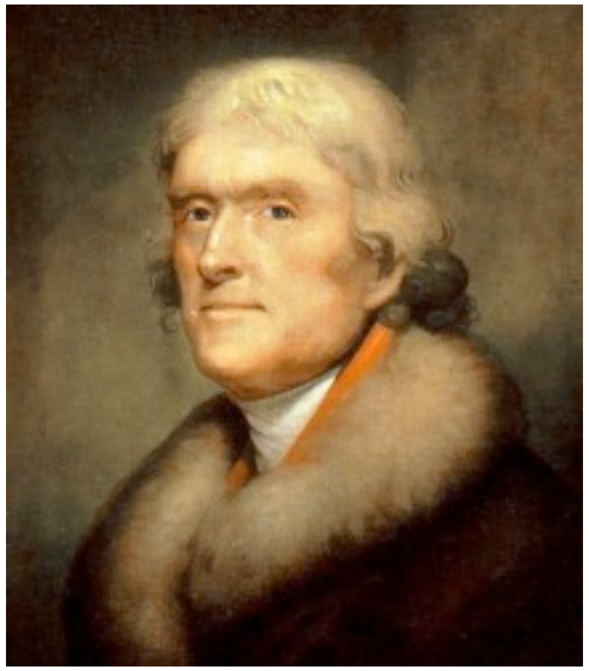
come at beginning of assembly file

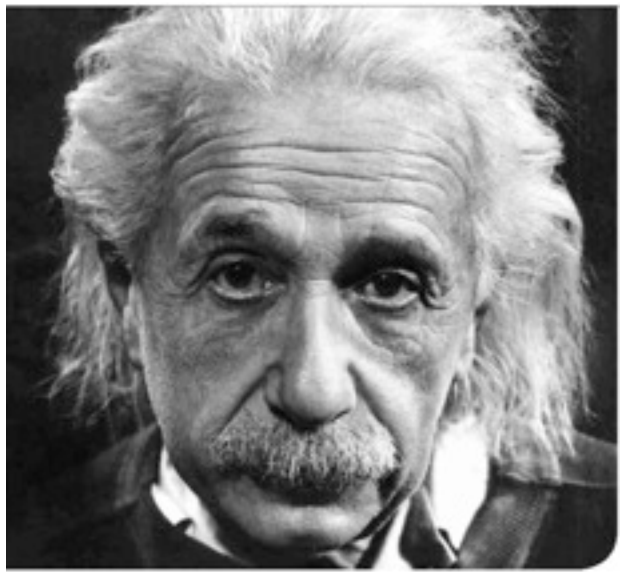
each can be referenced using

```
AREA sectionname{,attr}... ;AREA foo,CODE,READONLY (RO code section)
AREA sectionname{,attr}... ;AREA foo,DATA,READWRITE (RW data section)
; use the following with AREA
{label} SPACE expr ;foo SPACE 123 (123 bytes of zero---allocate space
                   ;for variable)
{label} DCB expr{,expr} ;foo DCB “foo”,0 (null terminated string)
{label} DCD expr{,expr} ;foo DCD 1,2,3 (three words containing 1,2,3)
EXPORT symbol [attr{,type{,set}}{,SIZE=n}] ;EXPORT foo [DATA, SIZE=4]
                                           ;(global variable)
name EQU expr{, type} ;foo EQU 2 (foo=2---use for constants)
ALIGN {expr{,offset{,pad{,padsize}}}} ;ALIGN 2 (align code/data to
                                      ;half-word)
ARM/THUMB/THUMBX ;following instructions are ARM/Thumb/Thumb2
; required lines
END ;assembly file must end with END
ENTRY ;if the program is called, it starts on this line
```

lingua franca

1. greek
2. arabic
3. latin
4. french
4.5 german
5. english
6. C-like*





*http://www.slate.com/articles/technology/technology/2012/01/learn_to_program_make_a_free_weekly_coding_lesson_your_new_year_s_resolution_.html

alignment

or what is boundary of piece of data/code?

how data/code is accessed, e.g.

`0x01234567`

tells the minimum size of data/code

alignment	data	addr (assume byte addressing)
byte	0x67 0x45 0x23 0x01	0x00 0x01 0x02 0x03
half-word	0x4567 0x0123	0x00 0x02
word	0x01234567 0x89ABCDEF	0x00 0x04

valid memory addr of this form

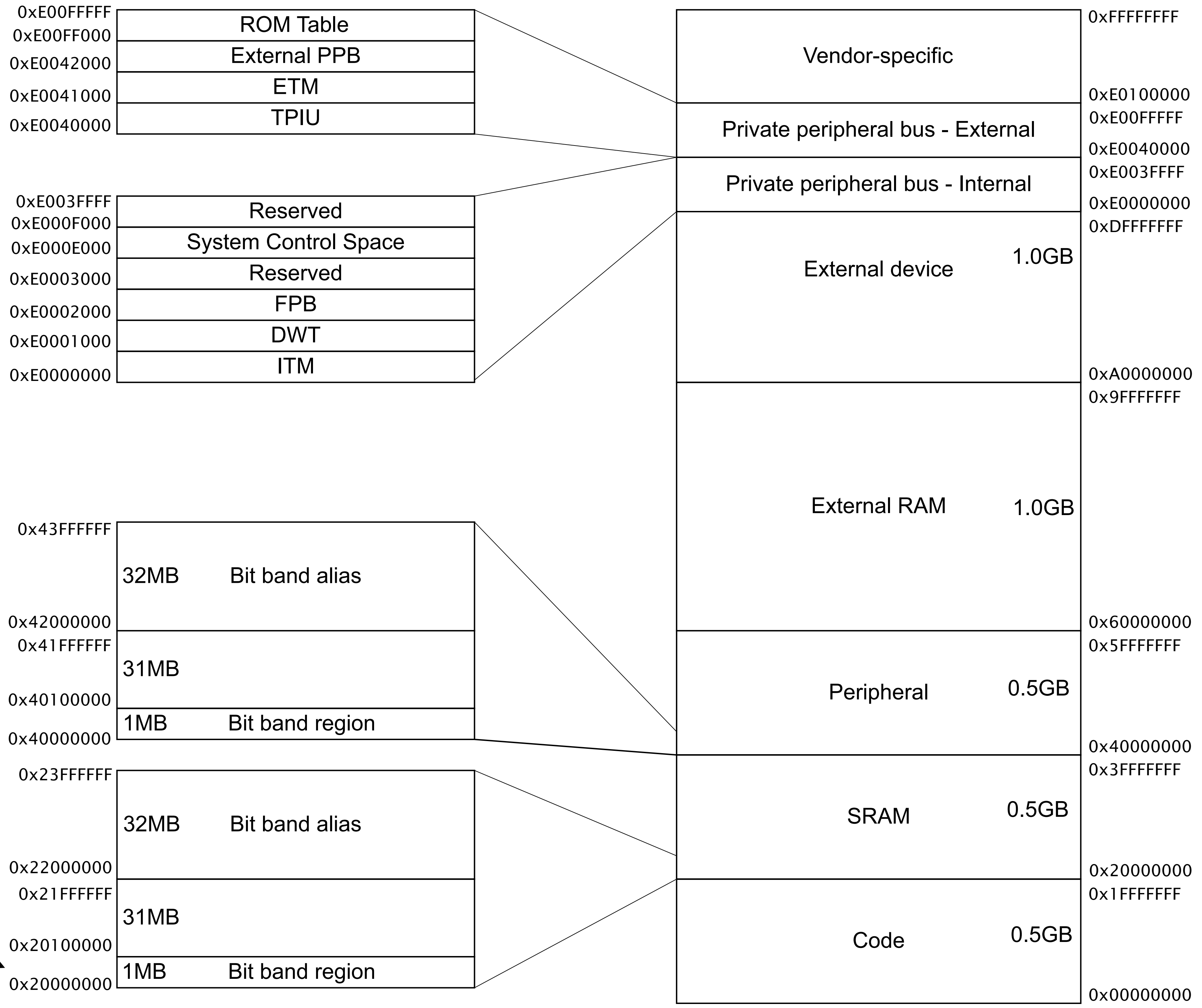
Cortex-M3 instructions are half-word aligned; LDR & STR can access byte addr

ARM memory map

which addresses point to which things

ARM has static memory map:

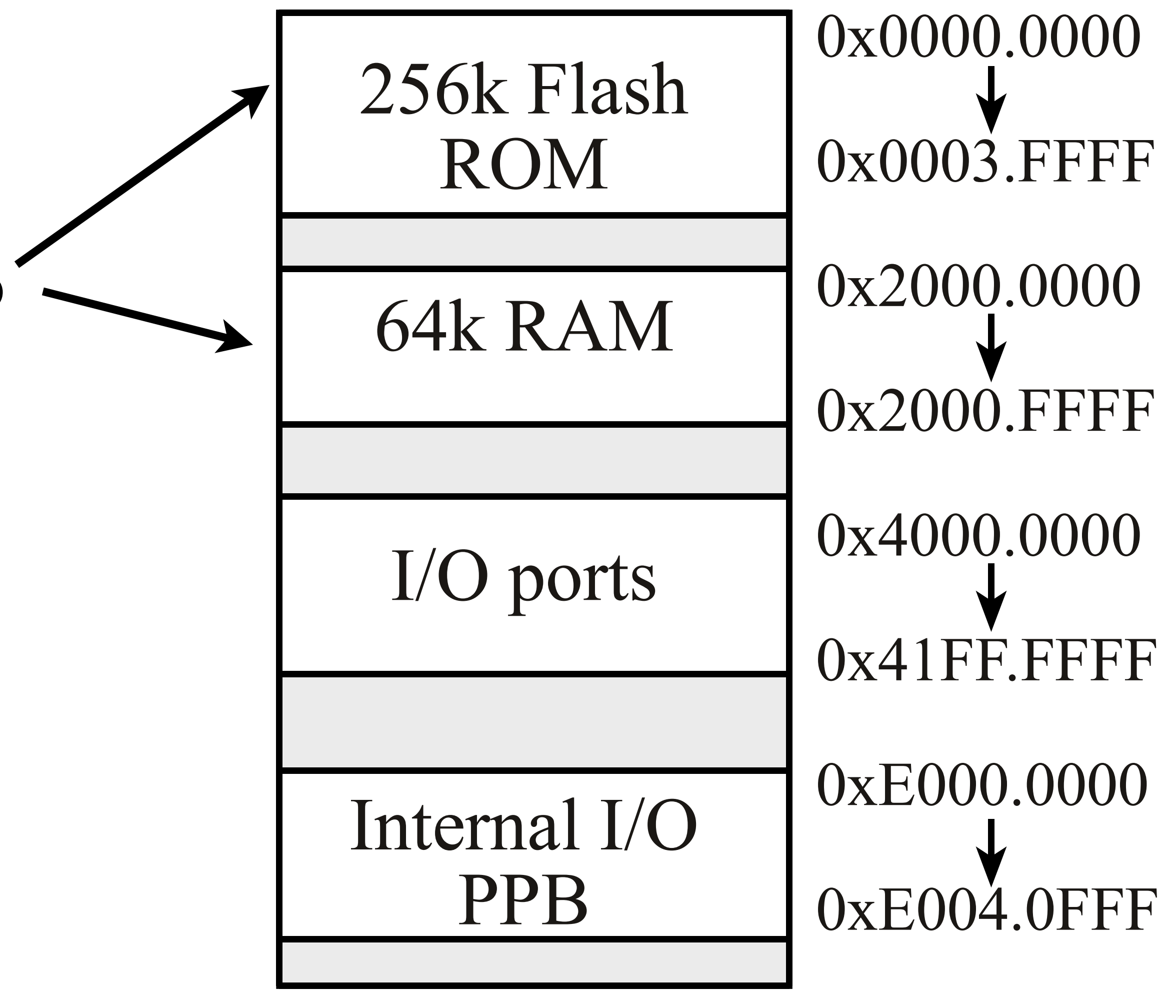
bit addressable (later)



memory map

static but devices don't have to use all:

< 0.5 GB allocated



TI LSM3S1968

unconditional branching

(a jump)

```
00000000: 6808          label1 LDR R0,[R1]
00000002: F1010104  ADD R1,#4
00000006: E7FB          B label1
```

goto label1

go +/- 32 MB
w/one branch

Q: branch to 0x12345678?

$\nearrow$

what about outside
of code space?

assembler directives
may help

how branching is achieved

(assume only 16-bit instructions)

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

this is an offset, relative to PC

assembler:

1. X:=addr. of `label`
2. Y:=addr. of `B`
3. offset = X-Y-4

(two's complement)

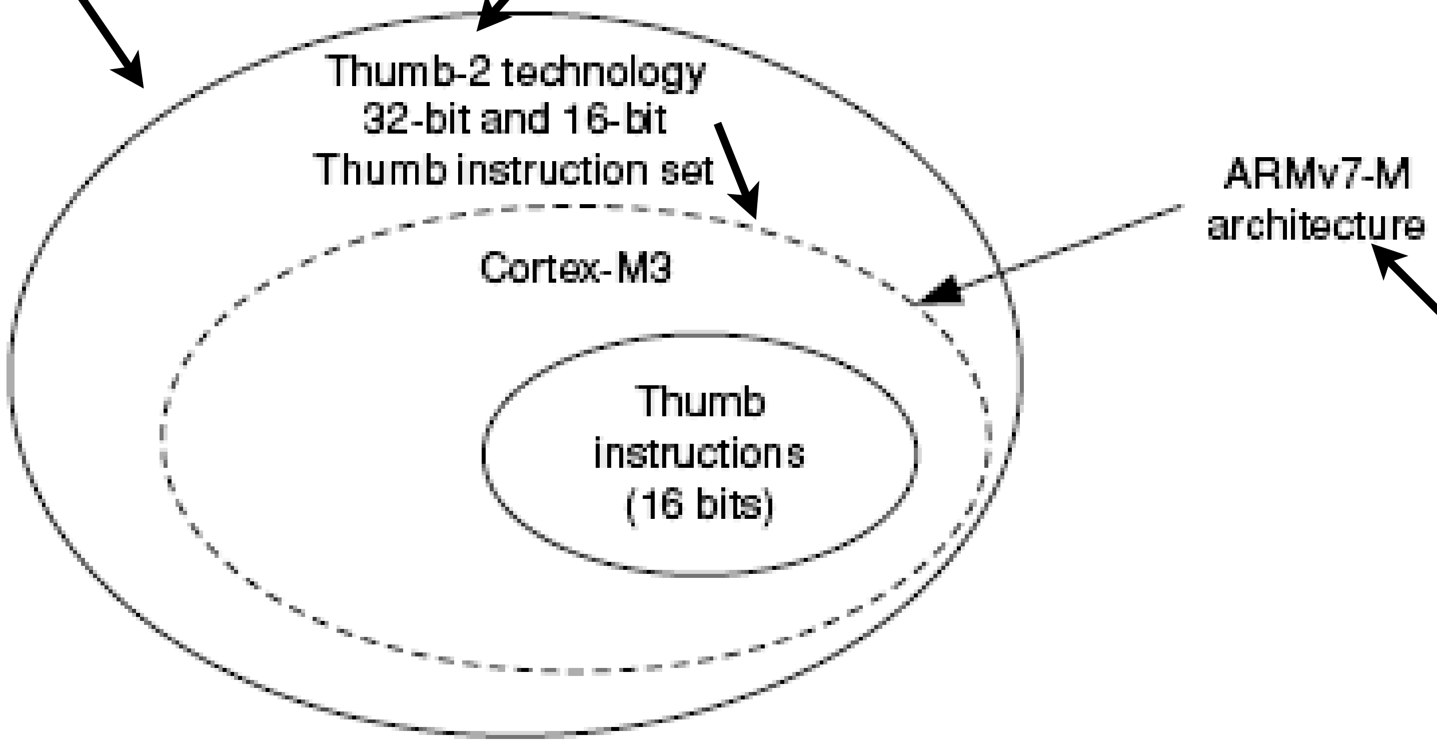
Thumb2: by the time branch can be executed PC has advanced by two 16-bit instructions

processor adds offset to PC

ARM, Thumb, and Thumb2

Thumb2 & ARM
(ARMv7)

basically ARM+Thumb
but not quite;
16/32-bit instructions;
still half-word aligned



product
differentiation

the ultimate guide to ARMv7-M (also the most painful) is the
ARMv7-M Architecture Reference Manual

compiler directives:

`THUMB` = Thumb2

`CODE16` = Thumb

your/lecture board (Cortex-M3) doesn't support `ARM`/`CODE32`

16-bit Thumb2 instruction encoding

assembly to bits

(what the assembler does)

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	0	0	0	Op		Offset5					Rs			Rd			***Move shifted register***
2	0	0	0	1	1	I	Op	Rn/offset3			Rs			Rd			***Add/subtract***
3	0	0	1	Op		Rd			Offset8								***Move/compare/add /subtract immediate***
4	0	1	0	0	0	0	Op				Rs			Rd			***ALU operations***
5	0	1	0	0	0	1	Op		H1	H2	Rs/Hs			Rd/Hd			***Hi register operations /branch exchange***
6	0	1	0	0	1	Rd			Word8								***PC-relative load***
7	0	1	0	1	L	B	0	Ro			Rb			Rd			***Load/store with register offset***
8	0	1	0	1	H	S	1	Ro			Rb			Rd			***Load/store sign-extended byte/halfword***
9	0	1	1	B	L	Offset5					Rb			Rd			***Load/store with immediate offset***
10	1	0	0	0	L	Offset5					Rb			Rd			***Load/store halfword***
11	1	0	0	1	L	Rd			Word8								***SP-relative load/store***
12	1	0	1	0	SP	Rd			Word8								***Load address***
13	1	0	1	1	0	0	0	0	S	SWord7							***Add offset to stack pointer***
14	1	0	1	1	L	1	0	R	Rlist								***Push/pop registers***
15	1	1	0	0	L	Rb			Rlist								***Multiple load/store***
16	1	1	0	1	Cond				Soffset8								***Conditional branch***
17	1	1	0	1	1	1	1	1	Value8								***Software Interrupt***
18	1	1	1	0	0	Offset11											***Unconditional branch***
19	1	1	1	1	H	Offset											***Long branch with link***
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	

src: THUMB Instruction Set

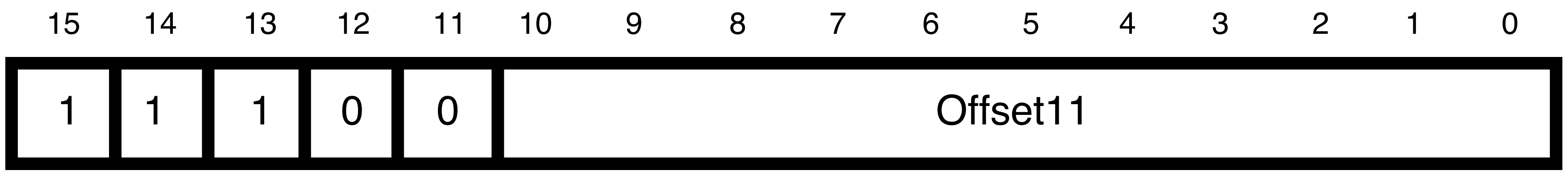
Thumb2 instructions: 16-bit

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

Q: how to get this from this

branch has this format

between label and branch



how do we calculate offset?

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

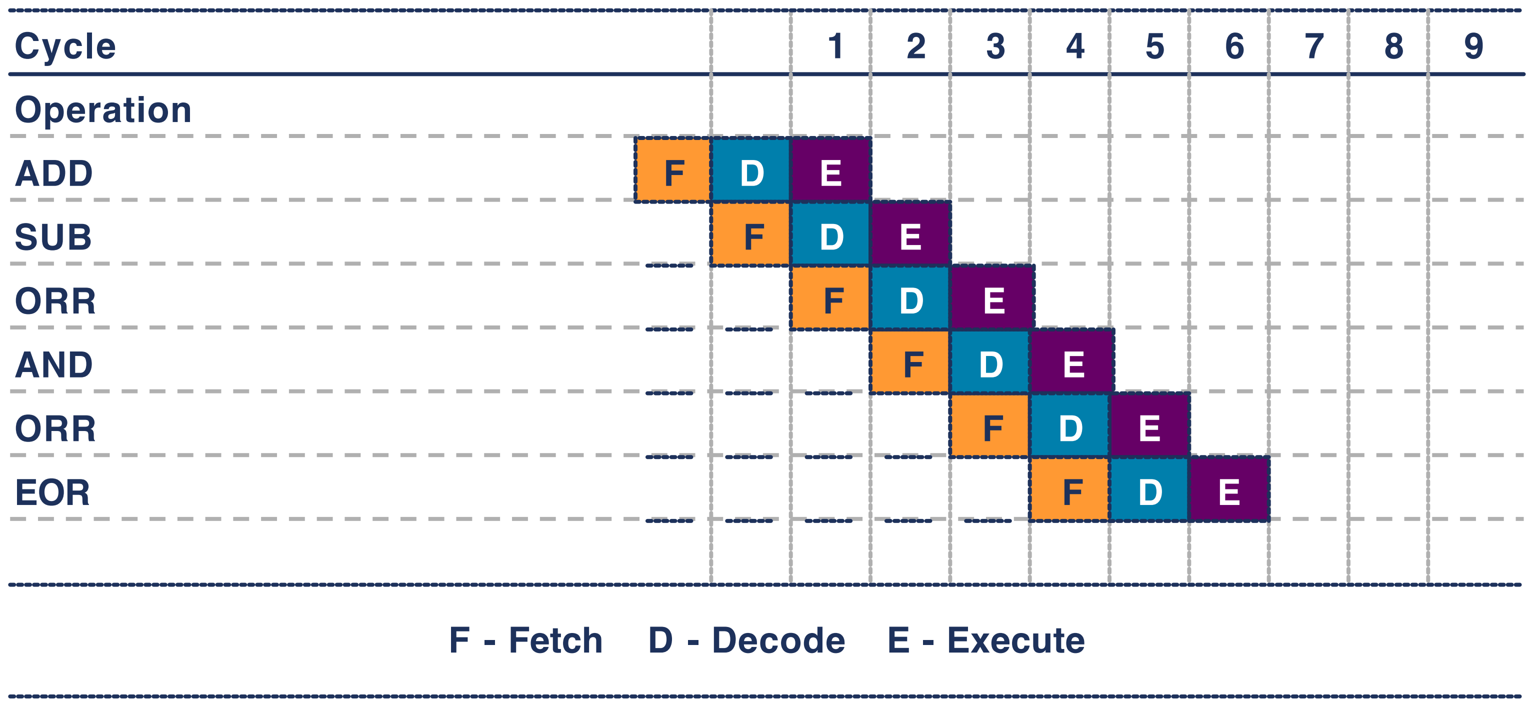
+1

0000 0000 1010

= -10

pipelining

three state pipeline:



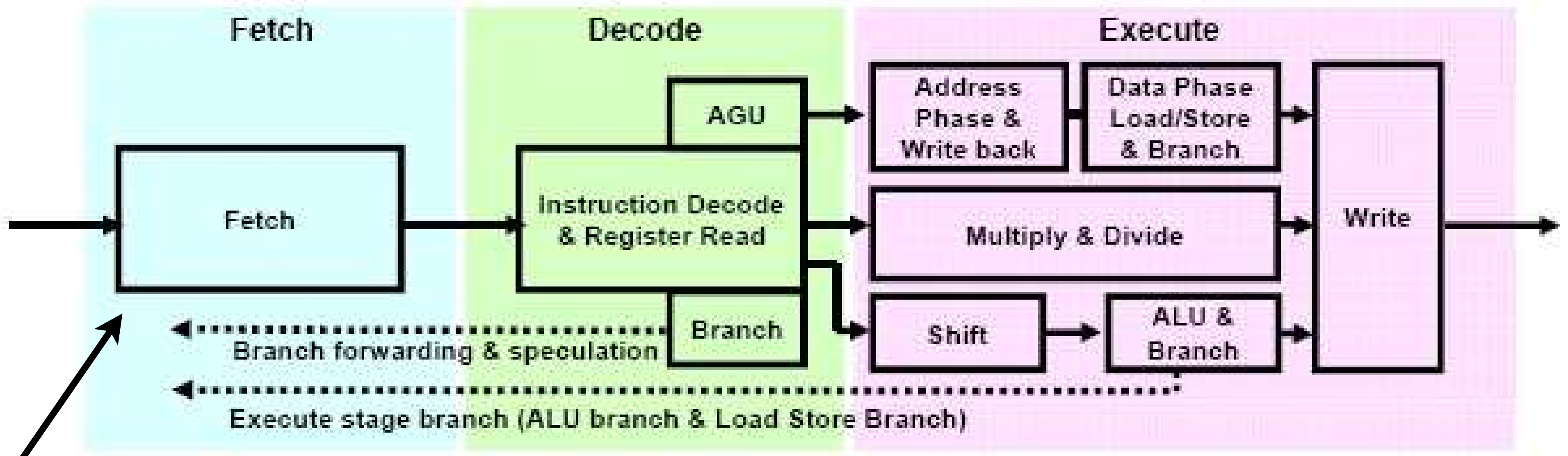
Cycle			1	2	3	4	5	6	7	8	9
Operation											
ADD	F	D	E								
SUB		F	D	E							
ORR			F	D	E						
AND				F	D	E					
ORR					F	D	E				
EOR						F	D	E			

F - Fetch D - Decode E - Execute

Thumb2 instructions: 16-bit

Cortex-M3 pipeline:

(three deep)

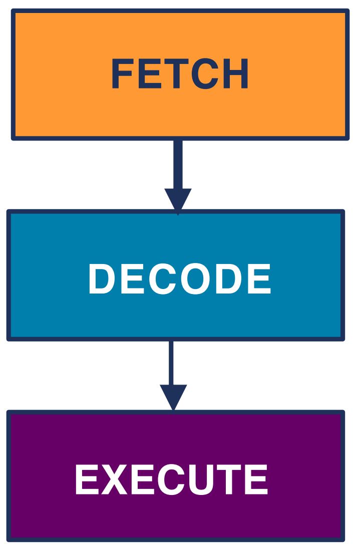

Fetch
Decode
Execute
Fetch
AGU
Instruction Decode & Register Read
Branch
Address Phase & Write back
Data Phase Load/Store & Branch
Multiply & Divide
Shift
ALU & Branch
Write
Branch forwarding & speculation
Execute stage branch (ALU branch & Load Store Branch)

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

Thumb2, too

inst. to be fetched

ARM	Thumb
PC	PC
PC - 4	PC-2
PC - 8	PC - 4



Instruction fetched from memory

Decoding of registers used in instruction

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