Assembly IV

ECE 3710

I went to a restaurant that serves "breakfast at any time". So I ordered French Toast during the Renaissance.

- Steven Wright

Q: c2asm

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

c2asm

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

```
mov R0,#12 ;h
mov R1,#0  ;i
mov R3,#34 ;k

iloop cmp R1,R0 ;i<h
      bge iloop_end
      mov R2,#0 ;j=0
jloop cmp R2,R1 ;j<i
      bge jloop_end
      add R3,#56  ;k+=56
      add R2,#1   ;j++
      b jloop
jloop_end add R1,#1 ;i++
          b iloop
iloop_end b iloop_end
```

using the stack:

0. SP points to top of stack

1. push:

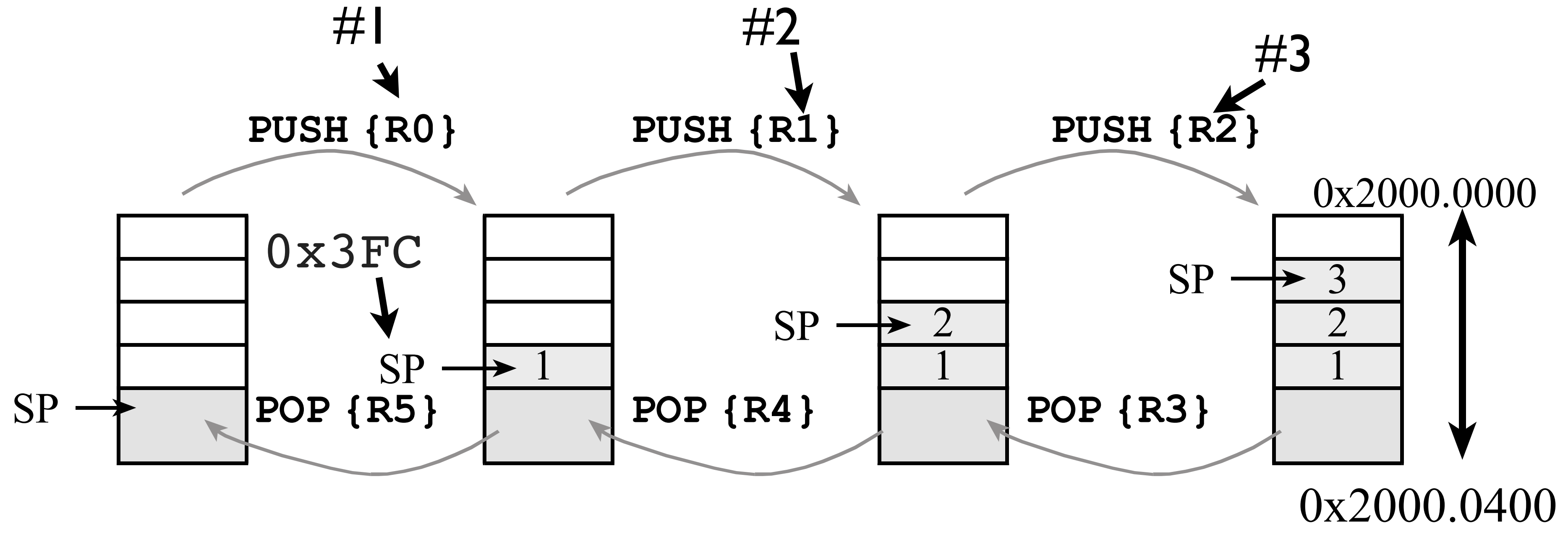
a. SP=SP-0x4

b. store word at SP

2. pop:

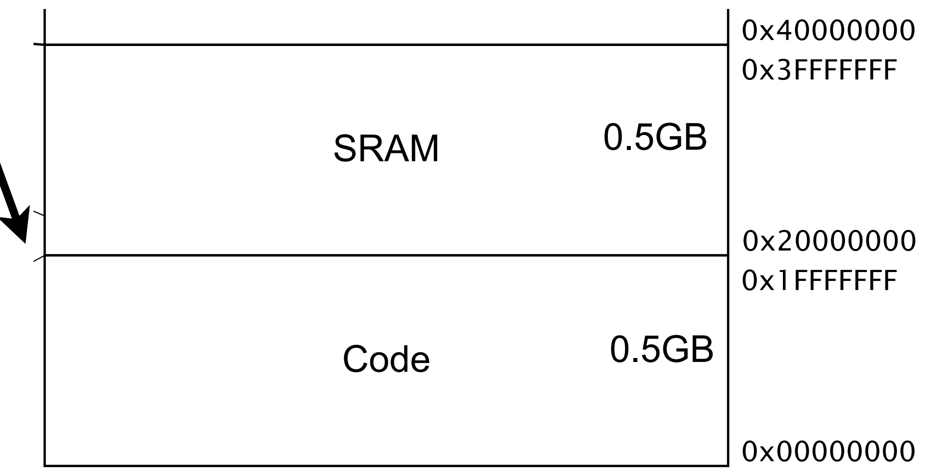
a. get word at SP

b. SP=SP+0x4



note: the stack grows down

where is the stack and how to access it?



the stack pointer (`SP`)

(special purpose register)

can be changed (hassle)

1. location: 0x20000000 (typical)
2. size is defined by you:

0x400 bytes

first entry in .asm

```
stack_size      EQU     0x400
                AREA    STACK, NOINIT, READWRITE
                ALIGN
stack_mem       SPACE   stack_size
initial_sp
```

what about multiple BLs?

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

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

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

LR=0x9

LR=0x17

no good, man

(we be stuck)

what about multiple `BL`s?

solution:

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

multiple BLs w/the stack

```
00000000 F04F 0000 Start mov,#0x0
00000004 F000 F803       BL add1      ← LR=0x9
00000008 F04F 000F       mov R0,#0xF
0000000C E7FE    loop    b loop
0000000E
0000000E B500    add1    push {LR}    ← LR=0x9
00000010 F100 0001       add R0,#1
00000014 F000 F803       BL add2      ← LR=0x19
00000018 F85D EB04       pop {LR}     ← LR=0x9
0000001C 4770            BX LR
0000001E B500    add2    push {LR}    ← LR=0x19
00000020 F100 0002       add R0,#2
00000024 F85D EB04       pop {LR}     ← LR=0x19
00000028 4770            BX LR
```

upon entering function: push LR to stack

before leaving function: pop LR from stack

multiple BLs w/the stack

```
00000000 F04F 0000 Start mov,#0x0
00000004 F000 F803       BL add1
00000008 F04F 000F       mov R0,#0xF
0000000C E7FE     loop   b loop
0000000E
0000000E B500     add1   push {LR}
00000010 F100 0001       add R0,#1
00000014 F000 F803       BL add2
00000018 F85D EB04       pop {LR}
0000001C 4770            BX LR
0000001E B500     add2   push {LR}
00000020 F100 0002       add R0,#2
00000024 F85D EB04       pop {LR}
00000028 4770            BX LR
```

LR=0x9

SP=0x...3FC
mem(0x...3FC)=0x9

LR=0x19

SP=0x...400
LR=0x9

SP=0x...3F8
mem(0x...3F8)=0x19

SP=0x...3FC
LR=0x19

```
Start mov,#0x0
  BL add1
  mov R0,#0xF
loop b loop
add1 push {LR}
  add R0,#1
  BL add2
  pop {LR}
  BX LR
add2 ...
  BX LR
```

Q: rearrange stack in `add2` so that `add1` returns to `add2`

logical instructions

```
mov r0,#0xff ;0b11111111
mov r1,#0xaa ;0b10101010
and r0,#0xaa ;r0 = 0xaa
orr r0,#0xff ;r0 = 0xff
eor r2,r0,r1 ;r2 = r0^r1 = 0b01010101
             ;            = 0x55
```

```
                add,orr,eor,bic,orn
               ↙
```

syntax: `<op>{s} {Rd,} Rn, <op2>`

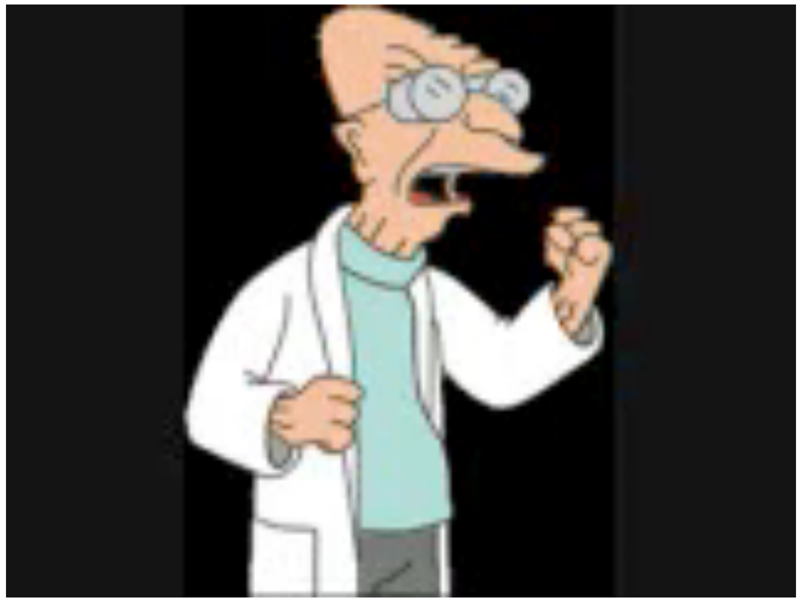
deciphering ARM syntax

many of you are wondering:

`<op>{s}{cond} ...`

what is this business?

to which many of your classmates respond:



...we want no more cryptic/obscure/ difficult instruction variations!

we know for b{cond} reads NZCV bits of APSR

`{COND}`

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)
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)	—

{s}

```
<op>{s}{cond} ...
```

append this
to instruction

sets one or more of `NZCV` in instruction dependent way

add/sub{s}

`instruction`	sets to one	if

`adds`	`N`
	`Z`
	`V`
	`C`
		result negative
		result is zero
		signed overflow
		unsigned overflow

`subs`	`N`
	`Z`
	`V`
	`C`
		result negative
		result is zero
		signed overflow
		unsigned overflow

syntax:

```
add/sub{s} {Rd,} Rn, <op2>
```

{s} example

```
     mov R0,#0xf
loop cmp R0,#0xf
     beq loop
```

equivalent

```
     mov R0,#0xf
loop subs R0,#0xf
     beq loop
```

sets Z=1

branch if

b{cond}

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

NOTE: `sleep(200)` doesn't put uC to sleep...

...it just loops and doesn't do anything useful for 200ms

Q: so how do we get delays?

Q: so how do we get delays?

R0>0x1; Z=0
R0=0x1; Z=1

A:

```
mov R0,#0xF
DELAY subs R0,#1
      bne DELAY
```

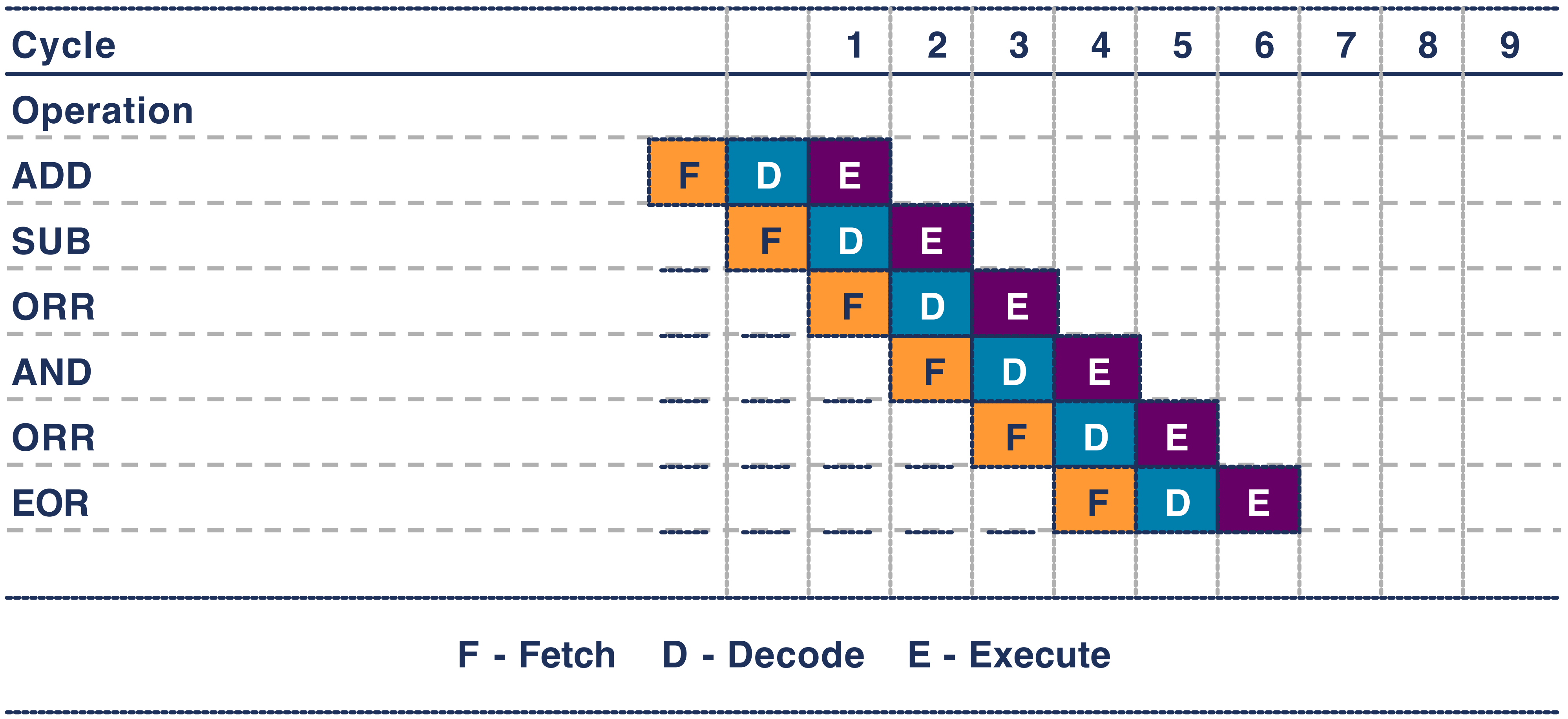
b if Z=0

Q: how many times do we execute this to get required delay?

A: depends on time per instruction

delay: how to make something fast, slow

machine cycle: time to finish one pipeline stage



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

if pipeline full:

then we only consider
machine cycles in execution stage
(fetch+decode is two cycles)

how long is a machine cycle?

$$\frac{\text{seconds}}{\text{machine cycle}} = \frac{\#\text{ clock cycles}}{\text{machine cycle}} \times \frac{\text{seconds}}{\#\text{clock cycles}}$$

Given uC with 11.0592 MHz clock and 1MC per 12 CC, one MC is:

$$\frac{12}{1} \times \frac{1}{11.0592 \times 10^6} = 1.085 \mu s$$

$$\frac{\text{seconds}}{\text{machine cycle}} = \frac{\text{\# clock cycles}}{\text{machine cycle}} \times \frac{\text{seconds}}{\text{\#clock cycles}}$$

how long (sec) is an instruction?

$$\frac{\text{seconds}}{\text{instruction}} = \frac{\#\text{ machine cycles}}{\text{instruction}} \times \frac{\text{seconds}}{\text{machine cycle}}$$

assume `MUL` is 4 MC, therefore `MUL` takes:

$$\frac{4}{1} \times \frac{1.085\mu s}{1} = 4.34\mu s$$

$$\frac{\text{seconds}}{\text{instruction}} = \frac{\text{\# machine cycles}}{\text{instruction}} \times \frac{\text{seconds}}{\text{machine cycle}}$$

```
mov R0,#0xFF
DELAY subs R0,#1
      bne DELAY
```

total time =

$$\# \text{ times instructions executed} \times \text{total MC of instructions} \times \frac{\text{seconds}}{\text{MC}}$$

$$= 256 \times (\underset{\substack{\uparrow \\ \text{MC of subs}}}{1} + \underset{\substack{\uparrow \\ \text{MC of bne}}}{2}) \times 1.085 = 833.28\mu s$$

increasing time of delay loops:

1. more iterations
2. run multiple times
3. add instructions to loop

(`NOP`, e.g.)

do nothing
(may be taken out by
assembler, though)

good news:

one clock cycle/per machine cycle

$$\uparrow$$

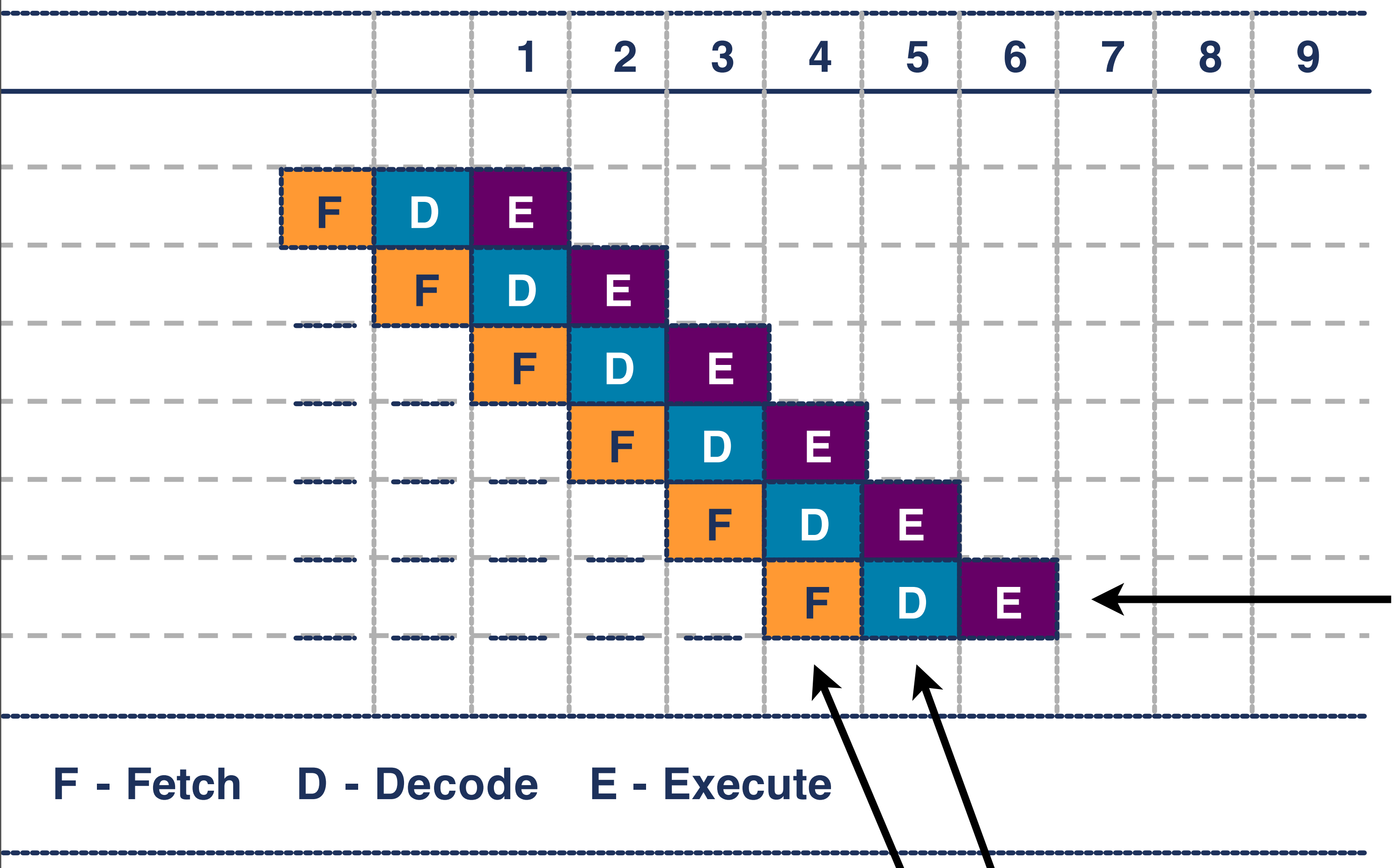
for Cortex-M3

bad news:

Instruction type	Size	Cycles count	Description
Data operations	16	1 (+P[a] if PC is destination)	ADC, ADD, AND, ASR, BIC, CMN, CMP, CPY, EOR, LSL, LSR, MOV, MUL, MVN, NEG, ORR, ROR, SBC, SUB, TST, REV, REV16, REVSH, SXTB, SXTH, UXTB, and UXTH. MUL is one cycle.
Branches	16	1+P[a]	B<cond>, B, BL, BX, and BLX. No BLX with immediate. If branch taken, pipeline reloads (two cycles are added).

a. Branches take one cycle for instruction and then pipeline reload for target instruction. Non-taken branches are 1 cycle total. Taken branches with an immediate are normally 1 cycle of pipeline reload (2 cycles total). Taken branches with register operand are normally 2 cycles of pipeline reload (3 cycles total). Pipeline reload is longer when branching to unaligned 32-bit instructions in addition to accesses to slower memory. A branch hint is emitted to the code bus that permits a slower system

b{cond}: one or two cycles added



branch here

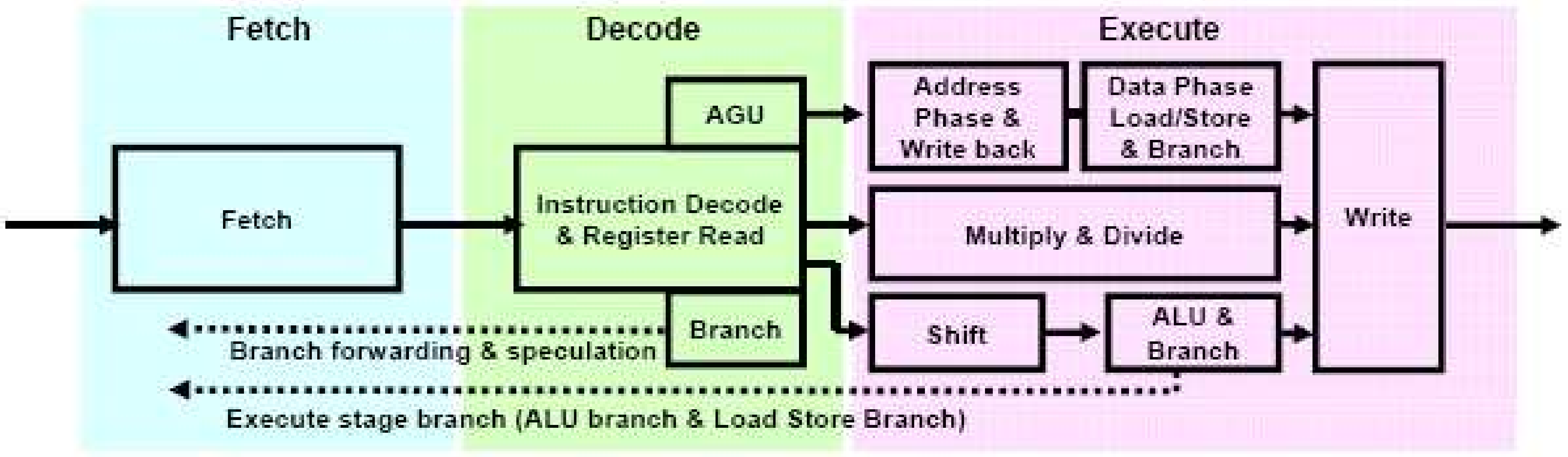
uC has loaded next two instructions after branch

but branch tells uC to load different instructions

(get rid of F&D)

this is what your program is
saying to the pipeline:

luckily...


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)

when decoding branch:

get fetch inst. for branch

$\swarrow$

if branch target 16-bit instruction:

not branch
has been assumed;
inst. fetched=>decoded

$\swarrow$

only flush decode

(b/c have branched inst)

takeaway:

1. ~2 MC per branch

2. loop calcs are approximations: need to measure loops using I/O pins and oscilloscope