EE 3171 Lecture 4

Assembly Language Programming

Lecture 4 Concepts

Building bigger assembly language programs

Learning more about the Cortex-M4 *instruction set architecture* (ISA)

Getting comfy with Keil µVision

Including debugging

Some *assembler directives*

A Programming Problem

In the last lecture, we wrote a teeny program to add 17 and 99.

Let’s look at that code again.

Zee Code

```
    THUMB
    AREA |.text|, CODE, READONLY, ALIGN=2
    EXPORT __main
__main
    MOV R6, #17
    ADD R7, R6, #99
Here B Here

Reset_Handler
    B __main

    ALIGN
    END
```

Both 17 and 99 are immediate values, which means they are constant values rather than variables.

A Programming Problem

In the last lecture, we wrote a teeny program to add 17 and 99.

What if these were the values of some variables instead?

Let `Op1 = 17` and `Op2 = 99`

Then, `Sum = Op1 + Op2`

Question 1: How do we create a variable?

Declaring Variables

```
 THUMB
 AREA DATA, READWRITE, ALIGN=2

 EXPORT Op1 [DATA,SIZE=4]
Op1 SPACE 4

 EXPORT Op2 [DATA,SIZE=4]
Op2 SPACE 4

 EXPORT Sum [DATA,SIZE=4]
Sum SPACE 4

 ALIGN
```

Variables go in the DATA area and we define it to be READWRITE so that we can perform both of those operations on values in that space.

Declaring Variables

```
 THUMB
 AREA DATA, READWRITE, ALIGN=2

 EXPORT Op1 [DATA,SIZE=4]
Op1 SPACE 4

 EXPORT Op2 [DATA,SIZE=4]
Op2 SPACE 4

 EXPORT Sum [DATA,SIZE=4]
Sum SPACE 4

 ALIGN
```

We've seen EXPORT before. It still advertises the label outside of the file, but this time, it's so we can watch that variable change in the Watch window.

Declaring Variables

```
    THUMB
    AREA DATA, READWRITE, ALIGN=2

    EXPORT Op1 [DATA,SIZE=4]
Op1 SPACE 4

    EXPORT Op2 [DATA,SIZE=4]
Op2 SPACE 4

    EXPORT Sum [DATA,SIZE=4]
Sum SPACE 4

    ALIGN
```

This is called a "label". It is used to give a symbolic name to a memory address.

Why? Three reasons:

- Variables
- Logically setting apart a section of code
- Targets for branch instructions.

Declaring Variables

```
 THUMB
 AREA DATA, READWRITE, ALIGN=2

 EXPORT Op1 [DATA,SIZE=4]
Op1 SPACE 4

 EXPORT Op2 [DATA,SIZE=4]
Op2 SPACE 4

 EXPORT Sum [DATA,SIZE=4]
Sum SPACE 4

 ALIGN
```

This is the assembler directive that sets aside a certain number of bytes of memory.

In this case, we reserve 4 bytes, or 1 word.

Declaring Variables

```
 THUMB
 AREA DATA, READWRITE, ALIGN=2

 EXPORT Op1 [DATA,SIZE=4]
Op1 SPACE 4

 EXPORT Op2 [DATA,SIZE=4]
Op2 SPACE 4

 EXPORT Sum [DATA,SIZE=4]
Sum SPACE 4

 ALIGN
```

And twice more, for the other operand and the result.

Finally, there's an `ALIGN`, because ARM seems to love that.

Initializing Variables

```
 EXPORT SystemInit
SystemInit
 LDR R4, =Op1
 MOV R5, #17
 STR R5, [R4]
 LDR R4, =Op2
 MOV R5, #99
 STR R5, [R4]
 MOV R4, #0
 MOV R5, #0
 BX LR
```

LDR is an instruction to LoaD a Register. The first parameter is the register to load.

There's a lot of good stuff here… So let's examine it.

Address vs. Value

Every variable in memory has both an *address* and a *value*.

It is ***crucial*** that you understand the difference between these characteristics.

In our example, Op1 has an *address* of `0x2000.000`. We want make its *value* 17.

Normally, we don't care what this number is.

To do this, we need to take two steps:

1. Get the address of that variable into a register.

2. Write the desired value to that address.

Initializing…

```
 EXPORT SystemInit
SystemInit
 LDR R4, =Op1
 MOV R5, #17
 STR R5, [R4]
 LDR R4, =Op2
 MOV R5, #99
 STR R5, [R4]
 MOV R4, #0
 MOV R5, #0
 BX LR
```

=Op1 is the syntax to get the address of a given label (read: variable).

Initializing…

```
 EXPORT SystemInit
SystemInit
 LDR R4, =Op1
 MOV R5, #17
 STR R5, [R4]
 LDR R4, =Op2
 MOV R5, #99
 STR R5, [R4]
 MOV R4, #0
 MOV R5, #0
 BX LR
```

STR is the instruction to write a value to memory (to STore from a Register).

The next bit is a little weird. You see a register name (R4) in square brackets ([ ]).

Indirect Addressing

The basic syntax here is: `[R`$_{base}$`, offset]`

The `offset` can be an immediate value or another register.

> There are actually lots of other offsets too, but these will suffice for us.

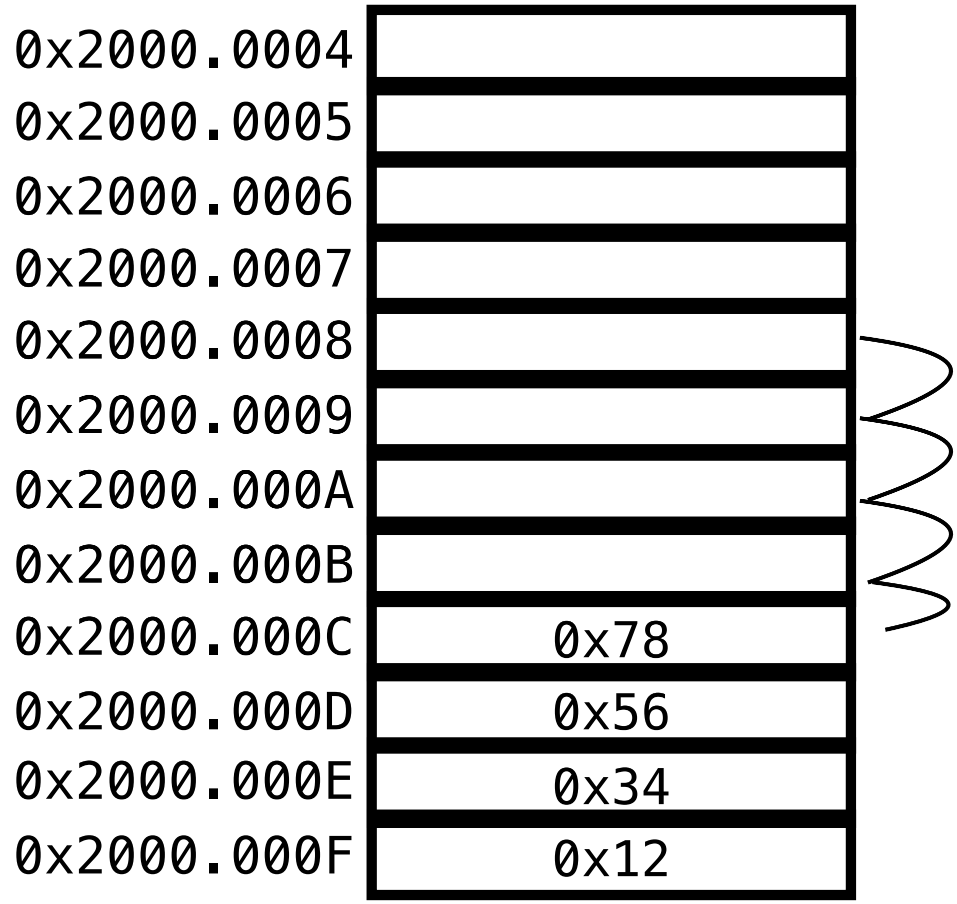
If you omit an `offset`, it is assumed to be zero.

Pictures!

Let's say:
`R3 = 0x2000.0008`,
`R5 = 0x12345678`
and we execute the instruction:
`STR R5, [R3, #4]`

Start at `0x2000.0008`, add 4, write the value there.

Confused about the data ordering? It's called *Little Endian*, and we won't worry about it too much.



Initializing…

```
 EXPORT SystemInit
SystemInit
 LDR R4, =Op1
 MOV R5, #17
 STR R5, [R4]
 LDR R4, =Op2
 MOV R5, #99
 STR R5, [R4]
 MOV R4, #0
 MOV R5, #0
 BX LR
```

So what does it all mean? Write the value of 17 to the address associated with the label Op1.

Initializing…

```
 EXPORT SystemInit
SystemInit
 LDR R4, =Op1
 MOV R5, #17
 STR R5, [R4]
 LDR R4, =Op2
 MOV R5, #99
 STR R5, [R4]
 MOV R4, #0
 MOV R5, #0
 BX LR
```

Down here we do the same thing to make `Op2 = 99`.

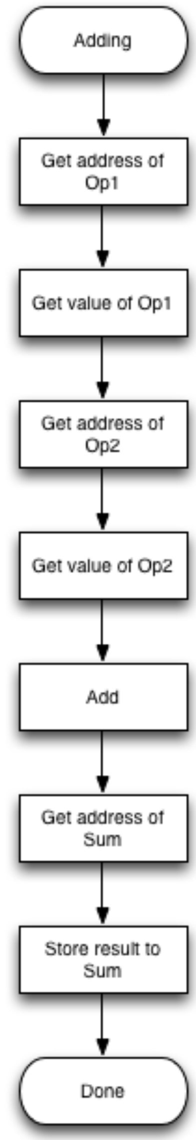
What about this? Ignore it for now.

Finally, we zero out everything we changed to leave no trace.

Step-by-Step

Can I just point out that that Wikipedia has a 10 MB image of a PB&J?

Flowing Through the Main Program



We need to think of this in a very step-by-step manner.

I need to get some addresses into registers so I can access the actual values.

Then I can do the adding.

Now let's look at the code.

Main Code

```
__main
 LDR R4, =Op1 ; Get a "pointer" to Op1
 LDR R5, =Op2 ; Get a "pointer" to Op2
 LDR R6, [R4] ; Get the actual Op1 into R6
 LDR R7, [R5] ; Get the actual Op2 into R7
 ADD R7, R6, R7 ; Add (Finally!)
 LDR R4, =Sum ; Get a pointer to Sum
 STR R7, [R4] ; Write the value to memory
Here B Here    ; Loop forever (for visibility)
```

I hope, at this point, there aren't any
major questions here.

General Data Processing Instructions

Arithmetic and Boolean Instructions:

`ADD`, `SDIV`, `UDIV`, `SUB`, `MUL`, `UMULL`, `AND`, `ORR`, `EOR`, `BIC` (bit clear)

Shift Instructions:

`ASR`, `LSL`, `LSR`, `ROR`

Comparisons:

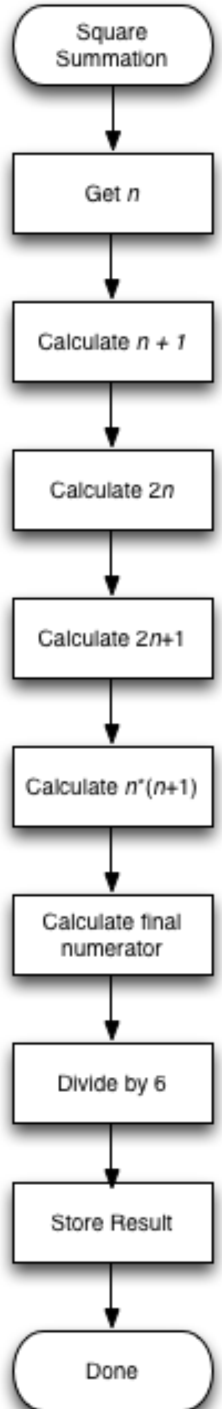
`CMP`, `TST`, `TEQ`

Sum of Squares

The summation of the first n integers squared is:

$$\sum_{k=0}^{n} = \frac{n(n+1)(2n+1)}{6}$$

Let's implement this in ARM assembly.



We'll get this ONCE and leave it in a register to reuse.

This will sit in a register, waiting.

This too.

Square Summation

```
 THUMB
 AREA DATA, READWRITE, ALIGN=2
 EXPORT n [DATA, SIZE=4]
n SPACE 4
 EXPORT Sum [DATA,SIZE=4]
Sum SPACE 4
 ALIGN
 AREA |.text|, CODE, READONLY, ALIGN=2
 EXPORT __main
__main
 LDR R6, =n       ; Get a pointer to n
 LDR R7, [R6]     ; Get value of n
 ADD R8, R7, #1   ; n+1
 MOV R9, #2
 MUL R9, R7       ; 2n
 ADD R9, R9, #1   ; 2n+1
 MUL R10, R7, R8; n*(n+1)
 MUL R10, R9      ; Last of the multiplying
 MOV R4, #6
 UDIV R10, R4     ; Divide
 LDR R6, =Sum
 STR R10, [R6]
Here B Here       ; Loop forever (for visibility)
```

Standard variable declarations, along with an EXPORT so Keil can show the values.

Square Summation

```
 THUMB
 AREA DATA, READWRITE, ALIGN=2
 EXPORT n [DATA, SIZE=4]
n SPACE 4
 EXPORT Sum [DATA,SIZE=4]
Sum SPACE 4
 ALIGN
 AREA |.text|, CODE, READONLY, ALIGN=2
 EXPORT __main
__main
 LDR R6, =n      ; Get a pointer to n
 LDR R7, [R6]    ; Get value of n
 ADD R8, R7, #1  ; n+1
 MOV R9, #2
 MUL R9, R7      ; 2n
 ADD R9, R9, #1  ; 2n+1
 MUL R10, R7, R8; n*(n+1)
 MUL R10, R9     ; Last of the multiplying
 MOV R4, #6
 UDIV R10, R4    ; Divide
 LDR R6, =Sum
 STR R10, [R6]
Here B Here      ; Loop forever (for visibility)
```

Pretty standard loading of a variable into a register.

Square Summation

```
 THUMB
 AREA DATA, READWRITE, ALIGN=2
 EXPORT n [DATA, SIZE=4]
n SPACE 4
 EXPORT Sum [DATA,SIZE=4]
Sum SPACE 4
 ALIGN
 AREA |.text|, CODE, READONLY, ALIGN=2
 EXPORT __main
__main
 LDR R6, =n     ; Get a pointer to n
 LDR R7, [R6]   ; Get value of n
 ADD R8, R7, #1 ; n+1
 MOV R9, #2
 MUL R9, R7     ; 2n
 ADD R9, R9, #1 ; 2n+1
 MUL R10, R7, R8; n*(n+1)
 MUL R10, R9    ; Last of the multiplying
 MOV R4, #6
 UDIV R10, R4   ; Divide
 LDR R6, =Sum
 STR R10, [R6]
Here B Here     ; Loop forever (for visibility)
```

Add 1 and put it in a different register so n is available for other things later.

Square Summation

```
 THUMB
 AREA DATA, READWRITE, ALIGN=2
 EXPORT n [DATA, SIZE=4]
n SPACE 4
 EXPORT Sum [DATA,SIZE=4]
Sum SPACE 4
 ALIGN
 AREA |.text|, CODE, READONLY, ALIGN=2
 EXPORT __main
__main
 LDR R6, =n     ; Get a pointer to n
 LDR R7, [R6]   ; Get value of n
 ADD R8, R7, #1 ; n+1
 MOV R9, #2
 MUL R9, R7     ; 2n
 ADD R9, R9, #1 ; 2n+1
 MUL R10, R7, R8; n*(n+1)
 MUL R10, R9    ; Last of the multiplying
 MOV R4, #6
 UDIV R10, R4   ; Divide
 LDR R6, =Sum
 STR R10, [R6]
Here B Here     ; Loop forever (for visibility)
```

The multiplication here is kind of weird. There's no immediate operand, so we have to load a register with the other operand.

Here's another weirdness: only two register operands. The MUL instruction will automatically write back to one of the source registers, if a third operand is omitted.

Square Summation

```
 THUMB
 AREA DATA, READWRITE, ALIGN=2
 EXPORT n [DATA, SIZE=4]
n SPACE 4
 EXPORT Sum [DATA,SIZE=4]
Sum SPACE 4
 ALIGN
 AREA |.text|, CODE, READONLY, ALIGN=2
 EXPORT __main
__main
 LDR R6, =n      ; Get a pointer to n
 LDR R7, [R6]    ; Get value of n
 ADD R8, R7, #1  ; n+1
 MOV R9, #2
 MUL R9, R7      ; 2n
 ADD R9, R9, #1  ; 2n+1
 MUL R10, R7, R8; n*(n+1)
 MUL R10, R9     ; Last of the multiplying
 MOV R4, #6
 UDIV R10, R4    ; Divide
 LDR R6, =Sum
 STR R10, [R6]
Here B Here      ; Loop forever (for visibility)
```

More of the same... multiplying with writeback.

Square Summation

```
 THUMB
 AREA DATA, READWRITE, ALIGN=2
 EXPORT n [DATA, SIZE=4]
n SPACE 4
 EXPORT Sum [DATA,SIZE=4]
Sum SPACE 4
 ALIGN
 AREA |.text|, CODE, READONLY, ALIGN=2
 EXPORT __main
__main
 LDR R6, =n      ; Get a pointer to n
 LDR R7, [R6]    ; Get value of n
 ADD R8, R7, #1  ; n+1
 MOV R9, #2
 MUL R9, R7      ; 2n
 ADD R9, R9, #1  ; 2n+1
 MUL R10, R7, R8; n*(n+1)
 MUL R10, R9     ; Last of the multiplying
 MOV R4, #6
 UDIV R10, R4    ; Divide
 LDR R6, =Sum
 STR R10, [R6]
Here B Here      ; Loop forever (for visibility)
```

The divide instruction has a lot in common with multiply.

Square Summation, *n*=5

```
 ADD R9, R9, #1 ; Now 2n+1
 MUL R10, R7, R8 ; n*(n-1)
 MUL R10, R9 ; Last of the multiplying
 MOVW R4, #6
 UDIV R10, R4 ; Divide
 LDR R6, =Sum
 STR R10, [R6]
Here B Here ; Loop forever (for visibility)

Reset_Handler
 B __main

 EXPORT SystemInit
SystemInit
 LDR R4, =n
 MOVW R5, #5
 STR R5, [R4]
 MOVW R4, #0
 MOVW R5, #0
 BX LR

 ALIGN
 END
```

Watch 1

Name	Value	Type
n	5	uint
Sum	55	uint

All is good.

Program Flow

```
1  THUMB
2  AREA DATA, READWRITE, ALIGN=2
3  EXPORT n [DATA, SIZE=4]
4 n SPACE 4
5  EXPORT Sum [DATA,SIZE=4]
6 Sum SPACE 4
7  ALIGN
8  AREA |.text|, CODE, READONLY, ALIGN=2
9  EXPORT __main
10 __main
11 LDR R6, =n      ; Get a pointer to n
12 LDR R7, [R6]    ; Get value of n
13 ADD R8, R7, #1  ; n+1
14 MOV R9, #2
15 MUL R9, R7      ; 2n
16 ADD R9, R9, #1  ; 2n+1
17 MUL R10, R7, R8; n*(n+1)
18 MUL R10, R9     ; Last of the multiplying
19 MOV R4, #6
20 UDIV R10, R4    ; Divide
21 LDR R6, =Sum
22 STR R10, [R6]
23 Here B Here     ; Loop forever (for visibility)
```

Starting at line 11, the program executes line-by-line, each in turn. It's a strictly linear program flow.

But what if we WANT non-linear flow?

Conditional Statements

Think about a simple `if`-`else` statement in C/C++:

```
if (myVariable == 1)
  anotherVariable = 2;
else
  anotherVariable = 3;
```

Let's break apart this code to get a clear idea of what's going on.

Conditional Statements

Think about a simple `if-else` statement in C/C++:

```
if (myVariable == 1)
  anotherVariable = 2;
else
  anotherVariable = 3;
```

This is a comparison that returns a yes/no answer.

Let's break apart this code to get a clear idea of what's going on.

Conditional Statements

Think about a simple `if`-`else` statement in C/C++:

```
if (myVariable == 1)
  anotherVariable = 2;
else
  anotherVariable = 3;
```

Do this ONLY if the answer is yes. Then, skip over the code associated with the "no" answer.

Let's break apart this code to get a clear idea of what's going on.

Conditional Statements

Think about a simple `if`-`else` statement in C/C++:

```
if (myVariable == 1)
  anotherVariable = 2;
else
  anotherVariable = 3;
```

Do this ONLY if the answer is no. Make sure the "yes" code doesn't get run.

Let's break apart this code to get a clear idea of what's going on.

Conditional Statements

Think about a simple `if`-`else` statement in C/C++:

```
if (myVariable == 1)
  anotherVariable = 2;
else
  anotherVariable = 3;
```

Pretend there's another line of code here. Regardless of whether the answer is "yes" or "no", we resume execution here.

Let's break apart this code to get a clear idea of what's going on.

Turning This Into Assembly

So, we need to do some sort of comparison that gives us a yes/no answer.

We need a way to skip the “yes” code if the answer is “no”.

And vice versa.

We need a way to bring everything back together.

Turning This Into Assembly

So, we need to do some sort of comparison that gives us a yes/no answer.

There are a couple ways to do this:
Explicit comparison instructions
Add a conditional suffix on to certain instructions.

We need a way to skip the “yes” code if the answer is “no”.

And vice versa.

We need a way to bring everything back together.

Turning This Into Assembly

So, we need to do some sort of comparison that gives us a yes/no answer.

There are a couple ways to do this:
Explicit comparison instructions
Add a conditional suffix on to certain instructions.

We need a way to skip the "yes" code if the answer is "no".

And vice versa.

This we do with BRANCH instructions.

We need a way to bring everything back together.

Turning This Into Assembly

There are a couple ways to do this:
Explicit comparison instructions
Add a conditional suffix on to certain instructions.

So, we need to do some sort of comparison that gives us a yes/no answer.

We need a way to skip the "yes" code if the answer is "no".

This we do with BRANCH instructions.

And vice versa.

We need a way to bring everything back together.

This we do with BRANCH instructions too.

The Simple Option: `IT`

There is an instruction that implements if-then statements in a completely intuitive fashion, but…

Can only handle a total of 4 (four) instructions.

So the `IT` instruction can't be your primary tool, but it's a good tool in the box.

Let's look at how it is used…

Using `IT`

Option 1: Take some action if a condition is true, do nothing otherwise.

Example: If the value in `R6` is greater than 65, `R7` should be set to 1. Otherwise, do nothing.

It would be great to start writing this code, but how do we know if `R6 > 65`?

Comparisons

The basic syntax is:
`CMP Rn, `*`Operand2`*

So sayeth the documentation: “These instructions compare the value in a register with *Operand2*. They update the condition flags on the result, but do not write the result to a register.”

Wait… condition flags?

And what’s that *`{cond}`* bit all about?

The Application PSR (**APSR**)

Program Status Register (PSR)

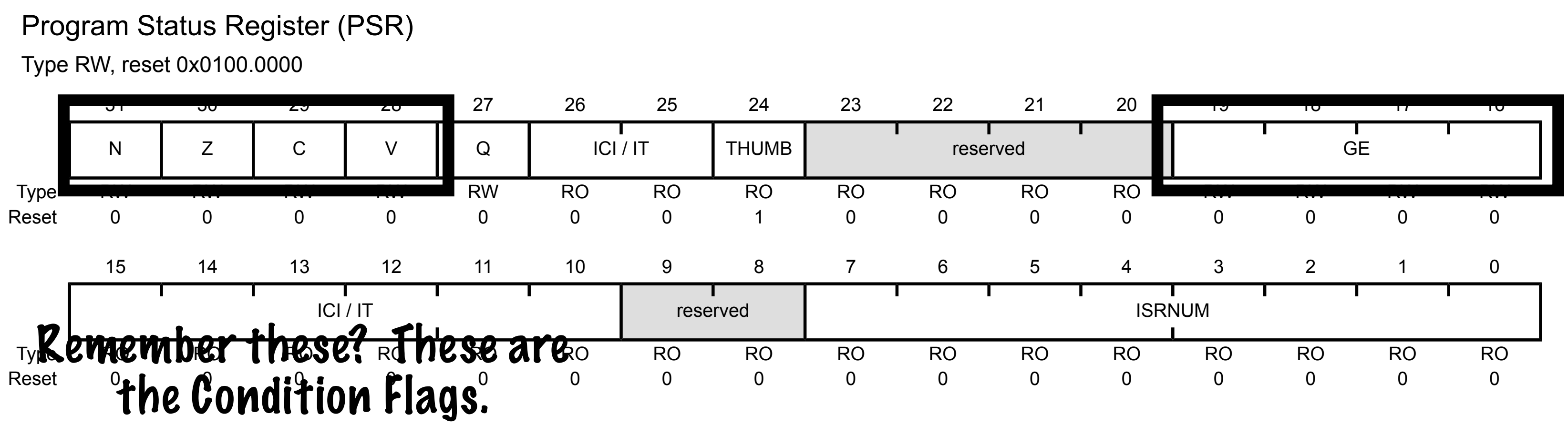
Type RW, reset 0x0100.0000

	[illegible]	[illegible]	[illegible]	[illegible]	27	26	25	24	23	22	21	20	[illegible]	[illegible]	[illegible]	[illegible]
	N	Z	C	V	Q	ICI / IT		THUMB	reserved				GE			
Type	[illegible]	[illegible]	[illegible]	[illegible]	RW	RO	RO	RO	RO	RO	RO	RO	[illegible]	[illegible]	[illegible]	[illegible]
Reset	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	ICI / IT						reserved		ISRNUM							
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

N - Set to a 1 if the previous operation result was negative or less than.
Z - Set to a 1 if the previous operation result was zero.
C - Set to a 1if the previous add resulted in a carry or the previous subtraction did *not* result in a borrow.
V - Set to a 1 if the previous operation resulted in overflow.

The Application PSR (**APSR**)



N - Set to a 1 if the previous operation result was negative or less than.
Z - Set to a 1 if the previous operation result was zero.
C - Set to a 1if the previous add resulted in a carry or the previous subtraction did *not* result in a borrow.
V - Set to a 1 if the previous operation resulted in overflow.

The Conditional Flags in Action

Let `R3 =17` and `R4 = 19`. What are the results from some comparisons?

`CMP R3, R4` -> `N = 1`, `Z = 0`, `V = 0`, `C = 0`

`CMP R3, R3` -> `N = 0`, `Z = 1`, `V = 0`, `C = 1`

`CMP R3, #0` -> `N = 0`, `Z = 0`, `V = 0`, `C = 1`

The Conditional Flags in Action

Let `R3 =17` and `R4 = 19`. What are the results from some comparisons?

`CMP R3, R4` -> `N = 1`, `Z = 0`, `V = 0`, `C = 0`

`CMP R3, R3` -> `N = 0`, `Z = 1`, `V = 0`, `C = 1`

N = 1 because 17-19 < 0.

`CMP R3, #0` -> `N = 0`, `Z = 0`, `V = 0`, `C = 1`

The Conditional Flags in Action

Let `R3 =17` and `R4 = 19`. What are the results from some comparisons?

`CMP R3, R4` -> `N = 1`, `Z = 0`, `V = 0`, `C = 0`

`CMP R3, R3` -> `N = 0`, `Z = 1`, `V = 0`, `C = 1`

`CMP R3, #0` -> `N = 0`, `Z = 0`, `V = 0`, `C = 1`

Z = 1 because 17-17 = 0.

The Conditional Flags in Action

Let `R3 =17` and `R4 = 19`. What are the results from some comparisons?

`CMP R3, R4` -> `N = 1`, `Z = 0`, `V = 0`, `C = 0`

`CMP R3, R3` -> `N = 0`, `Z = 1`, `V = 0`, `C = 1`

`CMP R3, #0` -> `N = 0`, `Z = 0`, `V = 0`, `C = 1`

`C = 1` because it requires a borrow to try and subtract 0 from 17.

Back to `IT`

Great. We can compare some value to some other value.

Now what?

Well, the syntax for the IT instruction says its format is: `IT{x{y{z}}} cond`

Whoa.
Well, let's start at the end.

{cond} Syntax

These are the different kinds of conditions the processor can detect.

They are based on a single PSR bit or a combination of 2 or 3 bits.

Suffix	**Flags**	**Meaning**
`EQ`	Z = 1	Equal
`NE`	Z = 0	Not equal
`CS or HS`	C = 1	Higher or same, unsigned
`CC or LO`	C = 0	Lower, unsigned
`MI`	N = 1	Negative
`PL`	N = 0	Positive or zero
`VS`	V = 1	Overflow
`VC`	V = 0	No overflow
`HI`	C = 1 and Z = 0	Higher, unsigned
`LS`	C = 0 or Z = 1	Lower or same, unsigned
`GE`	N = V	Greater than or equal, signed
`LT`	N != V	Less than, signed
`GT`	Z = 0 and N = V	Greater than, signed
`LE`	Z = 1 and N != V	Less than or equal, signed
`AL`	Can have any value	Always. This is the default when no suffix is specified.

Super Simple `IT`

Back to our example. If `R6 > 65`, `R7 ← 1`.

```
CMP R6, #65
 IT GT
 MOV R7, #1
```

But this doesn't build properly. "Flag preserving form of this instruction not available"?

Every instruction in the IT block must have a conditional suffix.

Super Simple `IT`

Back to our example. If `R6 > 65`, `R7` ← `1`.

```
CMP R6, #65
 IT GT
 MOVGT R7, #1
```

The sad part is that Keil won't do syntax coloring on `MOVGT`.

Not Everything Works with Just One Line

What if you need something more complex?

Not everything can be done in just one line of code. (Think about our three step process for writing a variable to memory.)

This is what that crazy `{x{y{z}}}` syntax is about. You can specify up to four instructions on the one comparison.

So let's extend our example. Instead of just changing registers, let's use some variables.

Longer `IT`

Let `speed` be the speed of a car and `tooFast` to be a variable that should be set to 1 if `speed` > 65.

```
LDR R6, =speed
LDR R7, [R6]
CMP R7, #65
ITTT GT                  # IF Greater Than
LDRGT R8, =tooFast # THEN
MOVGT R9, #1             # THEN
STRGT R9, [R8]           # THEN
```

Not Really Complete

That's not really complete code, is it?

We should really put a 0 into `tooFast` if `speed` <= 65.

The first thing is to take advantage of the `E` (that's the "else" clause).

So:

```
; insert previous code here
;
ITTTE GT            # IF Greater Than
LDRGT R8, =tooFast # THEN
MOVGT R9, #1       # THEN
STRGT R9, [R8]     # THEN
MOVLE R9, #0       # ELSE
```

But that's the four instructions.
We're out of space,
but not done.

A Way Around

Identify the code that is common to both cases and move it out of the `IT` block.

So:

```
; insert previous code here
;
LDR R8, =tooFast    # Common BEFORE the compare

CMP R7, #65
ITE GT              # IF Greater Than
MOVGT R9, #1        # THEN
MOVLE R9, #0        # ELSE

STR R9, [R8]        # Common AFTER the compare
```

More Complicated Logic

Sometimes an `IT` block won't do it.

> You can't express truly complicated logic in just 4 lines.

The solution is a set of instructions that modify the PC called *branches*.

Branches

Branches are like a fork in the road. If a condition is true, we go one way. If a condition is false, we go the other way.

Specifically, a branch is associated with a label. If the condition is true, we skip over any code between where we are and where the label exists in your code.

There are two basic kinds of branches: *conditional* and *unconditional*.

An unconditional branch <u>always</u> evaluates to true.

The Basic Pattern

The basic pattern you should usually follow is:

```
CMP <something, something>
B{cond} <TARGET>
```

E.g.,

```
CMP R3, #0
BEQ EqualToZero
```

Complexities

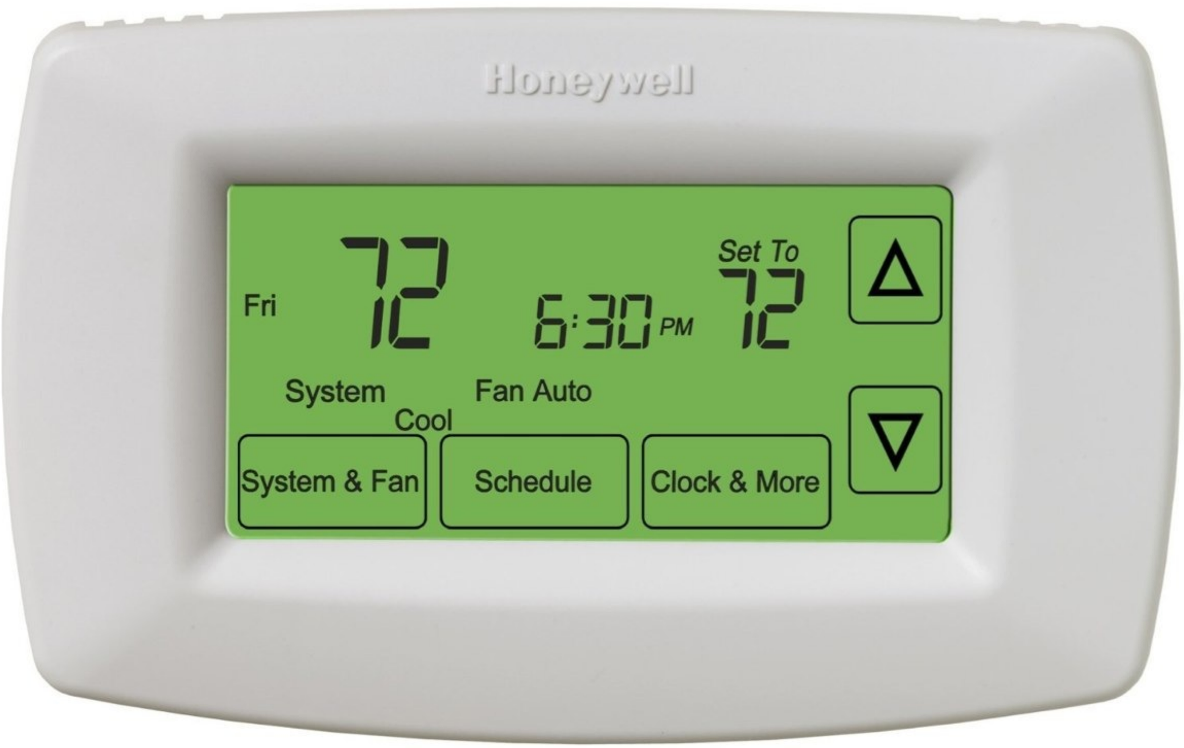
There are instructions like `CBNZ` that combine the comparison and branch into one instruction.

You can also add `s` to many instructions to get that operation to set the conditional flags.

For the sake of simplicity, just ignore these things and focus on the simple ways of doing them.

Example

Let there be a desired temperature. If it is too warm, we turn on the air conditioning. It it is too cold, we turn on the heat.



Thermometer Code

```
DesiredTemp EQU 68
CurrentTemp EQU 70
 AREA |.text|, CODE, READONLY, ALIGN=2
 EXPORT __main
__main
 MOV R4, #DesiredTemp
 MOV R5, #CurrentTemp
 CMP R4, R5
 BHI TurnOnHeat
TurnOnAC
 MOV R6, #2
 B MergePoint
TurnOnHeat
 MOV R6, #4
MergePoint
 B MergePoint

 ALIGN
 END
```

Thermometer Code

```
DesiredTemp EQU 68
CurrentTemp EQU 70
 AREA |.text|, CODE, READONLY, ALIGN=2
 EXPORT __main
__main
 MOV R4, #DesiredTemp
 MOV R5, #CurrentTemp
 CMP R4, R5
 BHI TurnOnHeat
TurnOnAC
 MOV R6, #2
 B MergePoint
TurnOnHeat
 MOV R6, #4
MergePoint
 B MergePoint

 ALIGN
 END
```

The EQU directive allows us to define constants.

Thermometer Code

```
DesiredTemp EQU 68
CurrentTemp EQU 70
 AREA |.text|, CODE, READONLY, ALIGN=2
 EXPORT __main
__main
 MOV R4, #DesiredTemp
 MOV R5, #CurrentTemp
 CMP R4, R5
 BHI TurnOnHeat
TurnOnAC
 MOV R6, #2
 B MergePoint
TurnOnHeat
 MOV R6, #4
MergePoint
 B MergePoint

 ALIGN
 END
```

Get the relevant temperatures.

Thermometer Code

```
DesiredTemp EQU 68
CurrentTemp EQU 70
 AREA |.text|, CODE, READONLY, ALIGN=2
 EXPORT __main
__main
 MOV R4, #DesiredTemp
 MOV R5, #CurrentTemp
 CMP R4, R5
 BHI TurnOnHeat
TurnOnAC
 MOV R6, #2
 B MergePoint
TurnOnHeat
 MOV R6, #4
MergePoint
 B MergePoint

 ALIGN
 END
```

Here's the comparison.

Thermometer Code

```
DesiredTemp EQU 68
CurrentTemp EQU 70
 AREA |.text|, CODE, READONLY, ALIGN=2
 EXPORT __main
__main
 MOV R4, #DesiredTemp
 MOV R5, #CurrentTemp
 CMP R4, R5
 BHI TurnOnHeat
TurnOnAC
 MOV R6, #2
 B MergePoint
TurnOnHeat
 MOV R6, #4
MergePoint
 B MergePoint

 ALIGN
 END
```

The BHI asks if R4 (first register) is greater than R5 (second register). If yes, jump to the label TurnOnHeat.

Because... the desired temperature is higher than the current temperature.

Thermometer Code

```
DesiredTemp EQU 68
CurrentTemp EQU 70
 AREA |.text|, CODE, READONLY, ALIGN=2
 EXPORT __main
__main
 MOV R4, #DesiredTemp
 MOV R5, #CurrentTemp
 CMP R4, R5
 BHI TurnOnHeat
TurnOnAC
 MOV R6, #2
 B MergePoint
TurnOnHeat
 MOV R6, #4
MergePoint
 B MergePoint

 ALIGN
 END
```

What about this? Well, any branch that evaluates to false ("no") is simply not taken and we fall through to the next line of code.

Thermometer Code

```
DesiredTemp EQU 68
CurrentTemp EQU 70
 AREA |.text|, CODE, READONLY, ALIGN=2
 EXPORT __main
__main
 MOV R4, #DesiredTemp
 MOV R5, #CurrentTemp
 CMP R4, R5
 BHI TurnOnHeat
TurnOnAC
 MOV R6, #2
 B MergePoint
TurnOnHeat
 MOV R6, #4
MergePoint
 B MergePoint

 ALIGN
 END
```

This is an example of an unconditional branch. It will ALWAYS be taken.

Thermometer Code

```
DesiredTemp EQU 68
CurrentTemp EQU 70
 AREA |.text|, CODE, READONLY, ALIGN=2
 EXPORT __main
__main
 MOV R4, #DesiredTemp
 MOV R5, #CurrentTemp
 CMP R4, R5
 BHI TurnOnHeat
TurnOnAC
 MOV R6, #2
 B MergePoint
TurnOnHeat
 MOV R6, #4
MergePoint
 B MergePoint

 ALIGN
 END
```

Why isn't there an unconditional branch here too? Because it's unnecessary.

The Unnecessary Branch

Why is that branch “unnecessary”?

Let’s say it was a conditional branch of some kind.

If it evaluates to true, we jump… to the next line of code.

If it evaluates to false, we fall through… to the *same* next line of code.

What if it was unconditional?

Then we always jump to the very same next line of code.

In other words, it’s existence *doesn’t matter*. So don’t put it in.

Example

Thermometers are programmable based on time. So let's look at the current time and set the desired temperature to different values depending on the time of day.

6AM - 8AM	8AM - 4PM	4PM - 11PM	11 PM - 6 AM
68^{o}F	60^{o}F	68^{o}F	65^{o}F

Variables and Constants

```
 THUMB
 AREA DATA, READWRITE, ALIGN=2
 EXPORT currTemp [DATA,SIZE=4]
currTemp SPACE 4
 EXPORT desiredTemp [DATA,SIZE=4]
desiredTemp SPACE 4
 EXPORT currTime [DATA,SIZE=4]
currTime SPACE 4 ; expressed as minutes past midnight
 ALIGN

; Constants
SixAM EQU 360
EightAM EQU 480
FourPM EQU 960
ElevenPM EQU 1380
```

Basic Flow

```
; Basic flow of the main program
; 1. Read the current time to find out what the desired temp is.
; 2. Store that value into the appropriate variable.
; 3. Check to see if the current temp is appropriate.
; 4. If not, turn on heat or AC as appropriate.
```

Putting these kinds of statements into the code as comments is called "self documenting code".

Preliminaries

```
 EXPORT __main
__main
 LDR R3, =desiredTemp
 LDR R1, =currTime
 LDR R2, [R1]
 CMP R2, #SixAM
 BLS TooEarly
```

Get pointers to the desired temperature and current time.

Then, get the actual time.

First Comparison

```
EXPORT __main
__main
 LDR R3, =desiredTemp
 LDR R1, =currTime
 LDR R2, [R1]
 CMP R2, #SixAM
 BLS TooEarly
```

Comparing with the lowest constant, `SixAM`. If it's less than that, it's just too early.

Seriously, people who get up before 6 AM are unnatural.

Handling the Rest of the First Comparison

```
AfterSixAM
 CMP R2, #EightAM
 BHI AfterEightAM

BetweenSixAndEight
 MOV R4, #68
 STR R4, [R3]
 B CheckTemp
```

We automatically fall through to this point if it's past 6 AM.

But now we have to check if the time is less than 8 AM as well.

If it is after 6, but before 8, the desired temperature is 68. Write that value to memory.

Remember that the address of `desiredTemp` is in R3.

More of the Same

```
AfterEightAM
 CMP R2, #FourPM
 BHI AfterFourPM

BetweenEightAndFour
 MOV R4, #60
 STR R4, [R3]
 B CheckTemp

AfterFourPM
 MOV R5, #ElevenPM
 CMP R2, R5
 BHI TooLate

BetweenFourAndEleven
 MOV R4, #68
 STR R4, [R3]
 B CheckTemp
```

Wrapping Things Up

```
TooEarly
 MOV R4, #65
 STR R4, [R3]
 B CheckTemp

TooLate
 MOV R4, #65
 STR R4, [R3]

CheckTemp
; Insert old themostat code here
 B CheckTemp
```

Shortcomings to This Code

It requires assembling the program to change any values.

We don't *actually* check the temperature or do anything about it.

Though I do have the "TODO" line in there.

Just being honest here.

Summing Up

We introduced a bunch of assembly language constructs, such as:

Assembler directives (`EQU`, `SPACE`, `AREA`, `ALIGN`, etc.)

Loading/storing registers

Some arithmetic

Comparisons, branches and `IT` blocks.