

# Loops and Addressing

Steven R. Bagley

# Introduction

- Started to look at writing ARM Assembly Language
- Saw the structure of various commands
- Looked at how we can put those commands together to form control structures seen in high-level languages (such as C)

```

        B main

menu    DEFB "G51CSA Vending Machine..\n\0"
coke    DEFB "Have a bottle of Coke\n\0"
error   DEFB "Incorrect option\n\0"
        ALIGN

main    SWI 1          ; Read a character
        CMP R0, #49
        BNE skip
        ADR R0, coke
        SWI 3
        B end
skip    ADR R0, error
        SWI 3
end     SWI 2

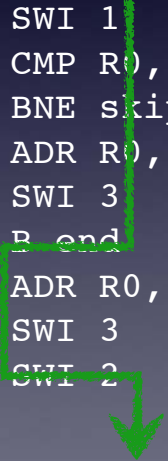
```

This is problematic, because the error message will always be printed

```
        B main

menu    DEFB "G51CSA Vending Machine..\n\0"
coke    DEFB "Have a bottle of Coke\n\0"
error   DEFB "Incorrect option\n\0"
        ALIGN

main    SWI 1          ; Read a character
        CMP R0, #49
        BNE skip
        ADR R0, coke
        SWI 3
        B end
skip    ADR R0, error
        SWI 3
end     SWI 2
```



This is problematic, because the error message will always be printed

```

        B main

menu    DEFB "G51CSA Vending Machine..\n\0"
coke    DEFB "Have a bottle of Coke\n\0"
error   DEFB "Incorrect option\n\0"
        ALIGN

main    SWI 1          ; Read a character
        CMP R0, #49
        DNE skip
        ADR R0, coke
        SWI 3
        B end
skip    ADR R0, error
        SWI 3
end     SWI 2

```

This is problematic, because the error message will always be printed

# Loops

- Important building blocks in programs
- `while` loops repeat a block of code whilst the condition holds
- But again, like `if`, ARM assembler does not provide a `while` loop
- Need to manufacture it out of simple components
- Already saw this early on in PRG...

# Loops using goto

- We can translate our `while` loop to using `gotos`
- Bad programming practice normally
- But a good halfway stage between structured programming and assembler
- Remember though that a `while` loop can execute zero or more times...

Run through an example in C

Show that they both run and produce the same result.

# while loop

```
while(i < 8)
{
    j = j + 3;
    i = i + 1;
}

goto while_cond;
while_loop:
    j = j + 3;
    i = i + 1;
while_cond:
    if(i < 8)
        goto while_loop;
```

Note how we've turned the structure of the while loop upside down

We've put the test at the end... There's reasons for this madness...

Can then convert this very easily into machine code

ASM version assumes i is in R1 and j is in R0



# while loop

```
        goto while_cond;
while_loop:
    j = j + 3;
    i = i + 1;
while_cond:
    if(i < 8)
        goto while_loop;
```

```
        B while_cond
while_loop
    ADD R0, R0, #3
    ADD R1, R1, #1
while_cond
    CMP R1, #8
    BLT while_loop
```

Note how we've turned the structure of the while loop upside down

We've put the test at the end... There's reasons for this madness...

Can then convert this very easily into machine code

ASM version assumes i is in R1 and j is in R0

# Worked Example

- Going to work through an example bit of code now
- To calculate the Highest Common Factor (HCF)
- Use Euclid's algorithm

# Euclid's HCF algorithm

- Euclid was a famous Greek mathematician (~300BC)
- Devised a simple algorithm for finding the Highest Common Factor of two integers

```
while(a != b)
{
    if(a > b)
        a = a - b;
    else
        b = b - a;
}
```

Let's convert this into assembler

# Loops

- Firstly, we need to remove the `while` loop
- Don't have such luxuries in assembly
- Only have branches, the equivalent of C's `goto`
- Rewrite our C program to use `ifs` and `goto` instead

Eek — GOTO alert... Go and do this, show the program works

# Registers

- Now we can start assigning the variables to registers
- Small number here, so we'll use `R0` to be equivalent to `a`, and `R1` to be equivalent to `b`
- Actual registers used doesn't matter (more or less)
- If you 'run out', you can always store the value in memory and the load it back into a register later

Okay now start developing an ARM version alongside the C version split screen  
then test in Komodo

# For Loops

- Again, we can model a `for` loop in assembler
- Already seen how to do this in G51PRG
- Saw that there is an equivalent `while` loop for every `for` loop
- So when we want to use a `for` loop we can convert it to a `while` loop
- Then convert the `while` loop to assembler as before

```
for(expr1; expr2; expr3)
    statement;

expr1;
while(expr2)
{
    statement;
    expr3;
}
```

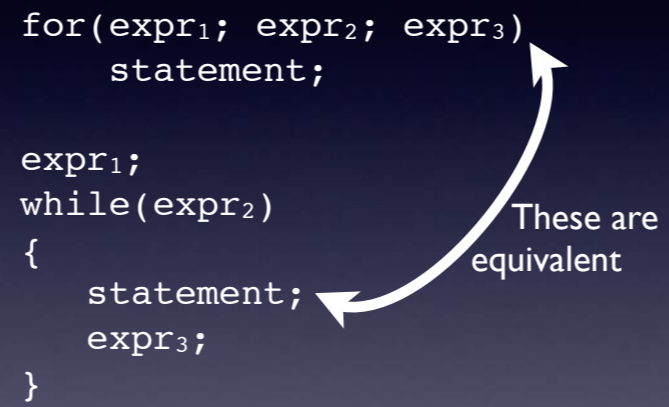
The equivalent while loop to any for loop

Go show how to rewrite the celsius program

```
for(expr1; expr2; expr3)
    statement;

expr1;
while(expr2)
{
    statement;
    expr3;
}
```

These are equivalent



The equivalent while loop to any for loop

Go show how to rewrite the celsius program



# while loop

```
for(i = 0; i < 10; i++)  
{  
    printf("%d\n", i);  
}  
  
i = 0;  
while(i < 10)  
{  
    printf("%d\n", i);  
    i++;  
}
```

Note how we've turned the structure of the while loop upside down

We've put the test at the end... There's reasons for this madness...

Can then convert this very easily into machine code

ASM version assumes i is in R1 and j is in R0

# while loop

```
i = 0;
while(i < 10)
{
    printf("%d\n", i);
    i++;
}

i = 0;
goto while_cond;
while_loop:
    printf("%d\n", i);
    i++;
while_cond:
    if(i < 10)
        goto while_loop;
```

Note how we've turned the structure of the while loop upside down

We've put the test at the end... There's reasons for this madness...

Can then convert this very easily into machine code

ASM version assumes i is in R1 and j is in R0

# Addressing

- Already seen how we can load and store values from memory

```
LDR R0, _a  
STR R1, _b
```

- But this can only load from a fixed address, and the same address every time
- Okay for global variables, wouldn't work for an array
- This is known as *Absolute Addressing*

Or at least it would be on most systems...

# Addressing on ARM

- ARM provides support for various ways to access memory
- At their heart, they are all just mechanisms for calculating the address to be accessed
- But provide very convenient mechanisms to handle it

# Handling Large Addresses

- ARM encodes all instructions in one 32-bit word
- This encodes the instruction and also what it operates on (registers, immediate values, etc)
- Different to a CISC chip which will use multiple words to encode the data
- Much faster to hide numeric constants and data addresses in the instruction themselves

ARM	68000
<code>ADR R0, label</code>	<code>lea label,a0</code>
<code>E24F0004</code>	<code>41F9</code> <code>0023</code> <code>3D56</code>

Comparison of ARM (RISC) and 68000 (CISC) encoding for a standard instruction to load an address into R0/A0. ARM implements it in one instruction read, while the 68000 requires three instruction reads.

# Addressing on ARM

- But the addresses are 32-bits wide...
- How does ARM hide them inside the the 32-bit instruction when there are only 8- or 12-bits spare?
- Uses a scheme that generates a 'useful subset' of all the 32-bit values
- Already seen this with immediate values in `MOV` instructions amongst others

# Immediate Value

- Encoded as an 8-bit value (0-255), and a 4-bit rotation value
- Rotation shifts the bits around the 32-bits of the register
- If value cannot be expressed using the above, we can always just load it from memory...



# Pseudo-Instruction

- Assembler recognises a pseudo-instruction which will automatically select whether to use a `MOV` or to load from memory  
`LDR R1, =0x12345678`
- Gets compiled to either a `MOV` instruction (if it will fit), or loads it from memory using `LDR`
- Assembler designates part of the program a 'literal pool' to store the constants
- Literal pool must be within  $\pm 4\text{KB}$  of the `PC`

But assembler handles it (go demo)

Pseudo-instruction because it doesn't exist — assembler generates a real instruction based on the program's context

# Using the PC to help

- Rotation isn't the only useful trick in generating a large constant (often intended as an address)
- Time to think about `ADR` in more detail
- Again, `ADR` is not an ARM instruction, but an assembler pseudo-instruction
- Assumes that the address to be loaded into a register is near the `PC`...

# ADR instruction

- If address is 'in range', then destination register can be set using a single-instruction by adding/ subtracting an offset to the value in PC
- Offset must be expressible in 8-bits with rotation
- So `ADR R0, one` might become  
`SUB R0, PC, #&18` or `ADD R0, PC, #&80`
- Depends on the relationship between the label and the instruction's location

# ADRL

- Not all address can be generated by this approach
- The offset must be expressible as a rotation of 8-bits
- Also have an `ADRL` pseudo-instruction
- This will use two (or more) instructions to generate the address...

Again usually `ADD` or `SUB` instructions

Given an example — use origin to demo it

# ADRL in aasm

- With `aasm`, `ADRL` will only generate a single instruction if possible
- Otherwise it may need two *or more* instructions
- Other assemblers may always generate two instructions
- The combination of PC-relative addressing with rotated constants is very powerful...

Can be important when writing code to know exactly how many instructions are executed...

```

alpha    ORIGIN 0x0100
         DEFW 20

beta     ORIGIN 0x1000
         DEFW 30

         ORIGIN 0x1010
         ENTRY

         ADRL R0, alpha    00001010 E24F0FC6 SUB R0 PC, #&318
                           00001014 E2400B03 SUB R0, R0, #&C00
         ADRL R0, beta     00001018 E24F0020 SUB R0, PC, #&20
         ADRL R0, gamma    0000101C E28F0EED ADD R0, PC, #&ED0
                           00001020 E2800A02 ADD R0, R0, #&2000

gamma    ORIGIN 0x3EF4
         DEFW 40

```

Various examples of ADRL in action in aasm

# Position Independence

- One other advantage of writing code like this is it is position-independent
- That is, providing all the access to addresses are relative to the PC then it doesn't matter where it is loaded in memory
- As soon as you hard-code an address (e.g. in a variable, or literal pool) then this breaks...

Doesn't mean the code cannot be loaded anywhere but the thing loading it will need to 'relocate' it to the new memory address...