Writing ARM Assembly

Steven R. Bagley

Introduction

- Previously, looked at how the system is built out of simple logic gates
- Last week, started to look at the CPU
- Writing code in ARM assembly language

Assembly Language

- CPUs consume instructions encoded in binary
  `1110 0000 1000 0000 0000 0000 0000 0001`
- This means add together R0 and R1 and store the result in R0
- Not very programmer friendly
- Not even if we view them in hex…

ARM encodes them in a fixed 32-bit word

Bit patterns not random controls the logic circuits in the CPU to perform the right operations

You don’t want to be so familiar you get jokes in the Star Trek episode titles…

Assembly Language

- CPUs consume instructions encoded in binary
  `0xE0800001`
- This means add together R0 and R1 and store the result in R0
- Not very programmer friendly
- Not even if we view them in hex…

ARM encodes them in a fixed 32-bit word

Assembly Language

- Programmers don't like binary machine code
- Computers don't understand English…
- Assembly Language is a compromise
- Gives English-like mnemonics for instructions
```
ADD R0, R0, R1
```
- Saves us memorising the bit patterns

Assembler

- Software that converts the mnemonics to binary
- Generates the correct bit patterns for each instruction
- But can also work out address offsets etc.
- And other conveniences…
- We are using `aasm` as our assembler
- Simpler programs than compilers

But will usually access it from within Komodo

So often have stricter input…

Assembly Syntax

- Assemblers tend to have more fixed syntax
- Although often simpler
- No functions
- Just instructions, or data
- Varies from assembler to assembler although often very similar, this is describes `aasm`

Assembly Syntax

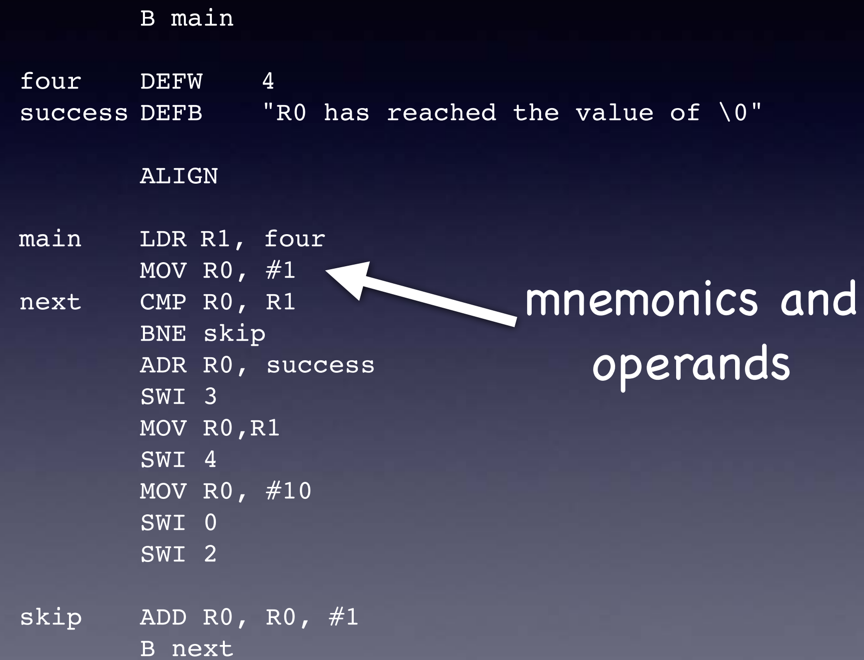
```
        B main

four    DEFW    4
success DEFB    "R0 has reached the value of \0"

        ALIGN

main    LDR R1, four
        MOV R0, #1
next    CMP R0, R1
        BNE skip
        ADR R0, success
        SWI 3
        MOV R0,R1
        SWI 4
        MOV R0, #10
        SWI 0
        SWI 2

skip    ADD R0, R0, #1
        B next
```

Mnemonics — symbol for the instruction (e.g. ADD, LDR, MOV etc) only ever one

Operands — what that instructions operates on (can be a variable number)

Although not enforced in aasm, they tend to be ‘tabbed’ in from the left hand column (using tabs or spaces) to allow room for the labels on the left.

Assembly Syntax



Mnemonics — symbol for the instruction (e.g. ADD, LDR, MOV etc) only ever one

Operands — what that instructions operates on (can be a variable number)

Although not enforced in aasm, they tend to be ‘tabbed’ in from the left hand column (using tabs or spaces) to allow room for the labels on the left.

Assembly Syntax

```
        B main

four    DEFW    4
success DEFB    "R0 has reached the value of \0"

        ALIGN

main    LDR R1, four
        MOV R0, #1
next    CMP R0, R1
        BNE skip
        ADR R0, success
        SWI 3
        MOV R0,R1
        SWI 4
        MOV R0, #10
        SWI 0
        SWI 2

skip    ADD R0, R0, #1
        B next
```

Mnemonics — symbol for the instruction (e.g. ADD, LDR, MOV etc) only ever one

Operands — what that instructions operates on (can be a variable number)

Although not enforced in aasm, they tend to be ‘tabbed’ in from the left hand column (using tabs or spaces) to allow room for the labels on the left.

Assembly Syntax

```
        B main

four    DEFW    4
success DEFB    "R0 has reached the value of \0"

        ALIGN

main    LDR R1, four
        MOV R0, #1
next    CMP R0, R1
        BNE skip
        ADR R0, success
        SWI 3
        MOV R0,R1
        SWI 4
        MOV R0, #10
        SWI 0
        SWI 2

skip    ADD R0, R0, #1
        B next
```

Labels just label things (instructions or data)

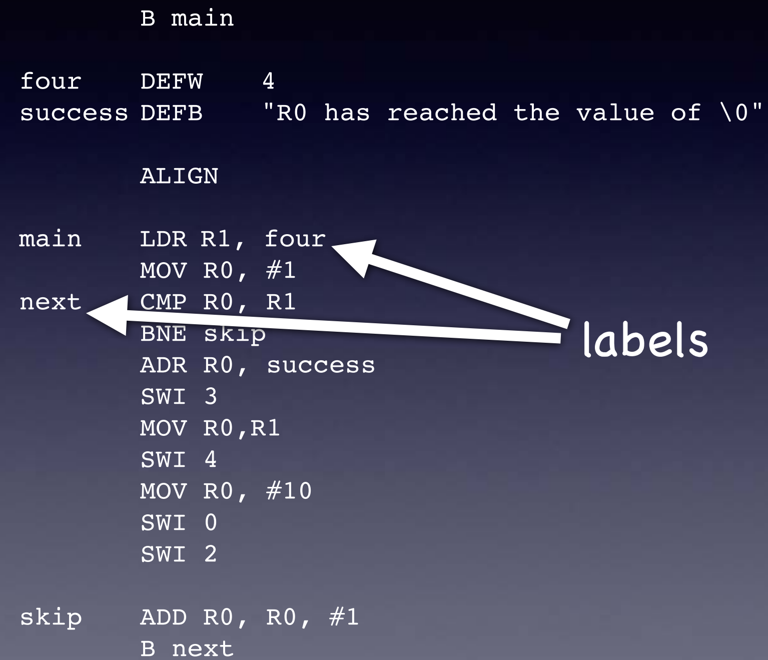
Can then be referred to elsewhere in the program by using the label name

Assembler automatically calculates the address and inserts it in the generated program (or calculates offsets when used with branches)

No need to declare them, assembler will make two-passes to work out where things are (a non-trivial process)

Must be the first thing on a line starting in the leftmost column

Assembly Syntax



Labels just label things (instructions or data)

Can then be referred to elsewhere in the program by using the label name

Assembler automatically calculates the address and inserts it in the generated program (or calculates offsets when used with branches)

No need to declare them, assembler will make two-passes to work out where things are (a non-trivial process)

Must be the first thing on a line starting in the leftmost column

Assembly Syntax

```
        B main

four    DEFW    4
success DEFB    "R0 has reached the value of \0"

        ALIGN

main    LDR R1, four
        MOV R0, #1
next    CMP R0, R1
        BNE skip
        ADR R0, success
        SWI 3
        MOV R0,R1
        SWI 4
        MOV R0, #10
        SWI 0
        SWI 2

skip    ADD R0, R0, #1
        B next
```

Labels just label things (instructions or data)

Can then be referred to elsewhere in the program by using the label name

Assembler automatically calculates the address and inserts it in the generated program (or calculates offsets when used with branches)

No need to declare them, assembler will make two-passes to work out where things are (a non-trivial process)

Must be the first thing on a line starting in the leftmost column

Assembly Syntax

```
        B main

four    DEFW    4
success DEFB    "R0 has reached the value of \0"

        ALIGN

main    LDR R1, four
        MOV R0, #1
next    CMP R0, R1
        BNE skip
        ADR R0, success
        SWI 3
        MOV R0,R1
        SWI 4
        MOV R0, #10
        SWI 0
        SWI 2

skip    ADD R0, R0, #1
        B next
```

Directives ‘direct’ the assembler to do things while generating code (e.g. ALIGN makes sure things align on a 4-byte boundary) — similar to #include/#define in C

Some do generate data into the code (e.g. DEFW)

Assembly Syntax

```
        B main

four    DEFW    4
success DEFB    "R0 has reached the value of \0"

        ALIGN

main    LDR R1, four
        MOV R0, #1
next    CMP R0, R1
        BNE skip
        ADR R0, success
        SWI 3
        MOV R0,R1
        SWI 4
        MOV R0, #10
        SWI 0
        SWI 2

skip    ADD R0, R0, #1
        B next
```

directives

Directives 'direct' the assembler to do things while generating code (e.g. ALIGN makes sure things align on a 4-byte boundary) — similar to #include/#define in C

Some do generate data into the code (e.g. DEFW)

Assembly Syntax

```
        B main

four    DEFW    4
success DEFB    "R0 has reached the value of \0"

        ALIGN

main    LDR R1, four
        MOV R0, #1
next    CMP R0, R1
        BNE skip
        ADR R0, success
        SWI 3
        MOV R0,R1
        SWI 4
        MOV R0, #10
        SWI 0
        SWI 2

skip    ADD R0, R0, #1
        B next
```

Directives 'direct' the assembler to do things while generating code (e.g. ALIGN makes sure things align on a 4-byte boundary) — similar to #include/#define in C

Some do generate data into the code (e.g. DEFW)

Assembly Directives

- Some useful directives:
  - `DEFW`, `DEFB` — define a word or byte respectively Causes the specified values to be inserted into the generated bitstream
  - Used here to store a string (but note we need to specify the NULL-character ourselves)

Unlike in C…

Assembly Directives

- Some useful directives:
  - `ALIGN` — align to a 4-byte boundary
  - `ORIGIN` — set address code is generated from
  - `EQU` — equate a name with something
    `fred EQU 42`
    This would mean we can use `fred` to mean 42 in our code

Equates are replaced at assembly time — like #defines are in C

Makes the code much more readable

Assembly Comments

- Can also add comments to our assembly
- A ‘;’ delimits the start and runs until end of line, e.g.
  `MOV R0, #65 ; moves 65 into R0`
- Probably more need for comments in assembly than C because the code is more cryptic

Hello World

```
         B main

hello    DEFB  “Hello World\n\0”
goodbye  DEFB  “Goodbye Universe\n\0”

         ALIGN

main     ADR   R0, hello    ; put address of hello string in R0
         SWI   3            ; print it out
         ADR   R0, goodbye  ; put address of goodbye string in R0
         SWI   3
         SWI   2            ; stop
```

Hello World program ARM style

Lets work through this bit by bit…

Branch to main, nothing special about the label could be anything

Hello World

```
         B fred

hello    DEFB  “Hello World\n\0”
goodbye  DEFB  “Goodbye Universe\n\0”

         ALIGN

fred     ADR   R0, hello    ; put address of hello string in R0
         SWI   3            ; print it out
         ADR   R0, goodbye  ; put address of goodbye string in R0
         SWI   3
         SWI   2            ; stop
```

even fred — it’s just a label…

Branch Instruction

- Mnemonic: `B`
- Causes execution to branch, or jump to a new location in memory
- Changes the `PC` register
- Takes one operand a 24-bit signed offset to the new location from this instruction
- Assembler calculates this offset for us automatically

Remember, PC is R15 on ARM

Offset multiplied by four (remember, each instruction is 4 bytes wide) and added to the PC (note the PC is always 8 bytes, 2 instructions, ahead of the one that is being executed)

Hello World

```
         B main

hello    DEFB  “Hello World\n\0”
goodbye  DEFB  “Goodbye Universe\n\0”

         ALIGN

main     ADR   R0, hello    ; put address of hello string in R0
         SWI   3            ; print it out
         ADR   R0, goodbye  ; put address of goodbye string in R0
         SWI   3
         SWI   2            ; stop
```

even fred — it’s just a label…

Hello World

```
         0xEA000007

hello    DEFB  “Hello World\n\0”
goodbye  DEFB  “Goodbye Universe\n\0”

         ALIGN

main     ADR   R0, hello    ; put address of hello string in R0
         SWI   3            ; print it out
         ADR   R0, goodbye  ; put address of goodbye string in R0
         SWI   3
         SWI   2            ; stop
```

even fred — it’s just a label…

Hello World

```
        B main

hello    DEFB  "Hello World\n\0"
goodbye  DEFB  "Goodbye Universe\n\0"

        ALIGN

main     ADR   R0, hello    ; put address of hello string in R0
         SWI   3            ; print it out
         ADR   R0, goodbye  ; put address of goodbye string in R0
         SWI   3
         SWI   2            ; stop
```

DEFB just defines the sequence of bytes for Hello world etc in the bitstream

ALIGN makes sure we are rounded to 4 bytes

ADR instruction

- Mnemonic: `ADR`
- Puts the address of a label in a register
- Two operands: register, and address
- This is not an instruction, but a convenience of the assembler
- Replaced by an addition/subtraction instruction based on the PC

Sometimes more than one instruction

Hello World

```
        B main

hello     DEFB  “Hello World\n\0”
goodbye   DEFB  “Goodbye Universe\n\0”

          ALIGN

main      ADR   R0, hello    ; put address of hello string in R0
          SWI   3            ; print it out
          ADR   R0, goodbye  ; put address of goodbye string in R0
          SWI   3
          SWI   2            ; stop
```

Software Interrupt

- Mnemonic: `SWI, SVC`
- Operand: 24-bit SWI number
- Generate a software interrupt…
- Causes the CPU to start executing from `0x8`
- Operand value used to decide what to do
- Trapped by the OS…

Has two opcodes because now referred to as a Service Call rather than a Software interrupt

Software Interrupt

- We don’t have an OS…
- But Komodo traps the SWIs for us
- Provides some useful I/O routines for us

Komodo-provided `SWIs`

SWI Number	Description
0	Outputs the character in the least significant byte of `R0` to the terminal window
1	Inputs the character typed into terminal window into the least significant byte of `R0`
2	Halts execution
3	Prints the string pointed to by `R0`
4	Print the integer value in `R0`

Note SWI 4 doesn't understand negative numbers!

No others are implemented

Hello World

```
        B main

hello     DEFB  "Hello World\n\0"
goodbye   DEFB  "Goodbye Universe\n\0"

          ALIGN

main      ADR   R0, hello    ; put address of hello string in R0
          SWI   3            ; print it out
          ADR   R0, goodbye  ; put address of goodbye string in R0
          SWI   3
          SWI   2            ; stop
```

Can now understand what this program does