DESN2000: Engineering Design & Professional Practice (EE&T)

Week 9

Constants, pseudo-instructions and literal pools

David Tsai

School of Electrical Engineering & Telecommunications

Graduate School of Biomedical Engineering

d.tsai@unsw.edu.au

Biomedical Microsystems Lab

This week

- Constants
- Pseudo-instructions
- Loading constants
- Loading addresses
- Assembler & linker

Constants

- ARM instructions are 32 bits long.
- How to fit a 32-bit constant into an instruction?

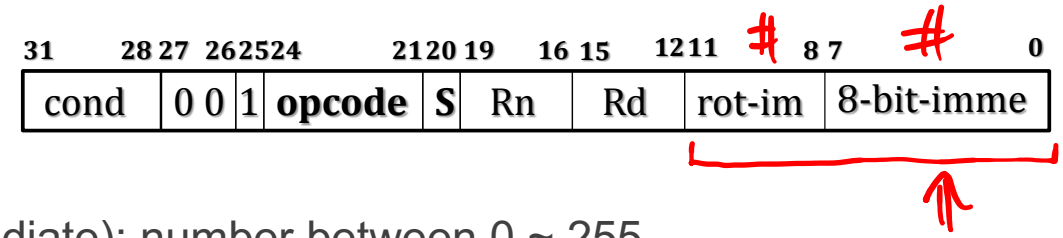
- MOV as example:

31–28	27–26	25	24–21	20	19–16	15–12	11–0
cond	0 0	1	**opcode**	**S**	Rn	Rd	shifter_operand

  - Bits[27:25] type of data processing instruction
  - Bits[24:21] opcode (1101 for MOV)
  - Bits[11:0] depends on addressing mode:
    - Register
    - Register with shift or rotate
    - Immediate

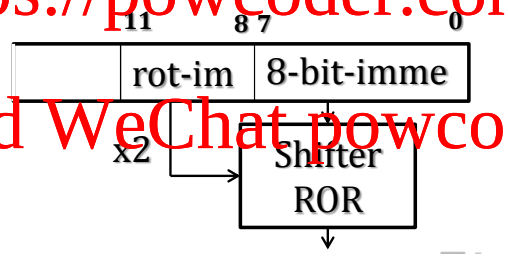
Constants

- Loading a constant, so immediate addressing mode format:



- Bits[7:0] (immediate): number between 0 ~ 255.

- Bits[11:8]: a rotation value, which is multiplied by 2, then used to rotate the forgoing 8-bit immediate via the ALU.



- Allows ARM to generate constants of the form:

$$v_{imme} \times \mathrm{ROR}(2v_{rot}) \quad \text{where} \quad v_{imme} \in [0:255]$$
$$v_{rot} \in [0:15]$$

- Doesn't cover all 32-bit integers.

Constants

- Example: calculate the rotation needed to generate the value 4080.
  - 4080 = $(1111\ 1111\ 0000)_2$ ← 4 pos shift
  - $(1111\ 1111)_2$, or 0xFF, can be rotated left by 4 bits to generate 4080.
  - Rotating left by 4 equivalent to rotating right by 28.
  - So `MOV R0, #4080` replaced by `MOV R0, #0xFF, ROR 28`.
  - Assembler does this for you.

Constants

- MVN (move negative) moves one's complement of the operand.
- We can generate additional constants using MVN with the rotation scheme:

$$\sim \left(v_{imme} \times \mathrm{ROR}(2v_{rot})\right) \quad \text{where} \quad v_{imme} \in [0:255]$$
$$v_{rot} \in [0:15]$$

ROR
LSL
LSR
ASR

- Example: loading the decimal 16777215 into R0.
  - 16777215 = 0x00FFFFFF
  - Rotating 0xFF right by 8 bits, then taking 1's complement.
  - So `MOV R0, #16777215` replaced by `MVN R0, #0xFF, ROR 8`.

- Bottom line: **MOV/MVN + rotation can only create a subset of 32-bit integers.**
  … also mentally inconvenient.

Constants

- More problems… we often do something like:

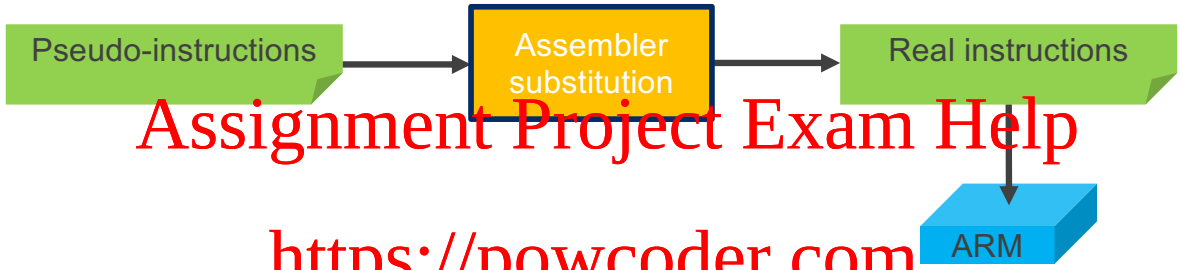
```
SRAM_BASE   EQU  0x04000000
            AREA EXAMPLE, CODE, READONLY
            ENTRY
            MOV  r0, #SRAM_BASE
            ...
```

All occurrences of "SRM_BASE" is replaced by "0x04000000"

- The code fails if SRAM_BASE changes to a value that cannot be generated by the byte rotation scheme.

Pseudo-instructions

- Solution: **pseudo-instructions**.
- Understood by the assembler (software) and replaced with real instructions for the ARM processor (hardware).



- HW is constrained by implementation considerations / limitations. Using software to work around these limitations.

- Example pseudo-instructions:
  - `LDR <rd>, =<constant>`
    Loads any numerical constant *const* into register *rd*.
  - `NOP`
    Do-nothing instruction, stand-in for real instruction `MOV R0, R0`.
  - `MOV R0, #0xFFFFFFFF`
    is replaced by real instruction `MVN R0, #0`.

Loading constants: LDR

- On encountering `LDR <rd>, =<constant>`, the assembler:
  1. Tries to substitute with **MOV / MVN, with optional rotation**.
  2. If this fails, create the *constant* in program space, known as a **literal pool**, then use the `LDR <rd>, [PC <offset>]` instruction to load this *constant*.

- Example demonstrating all 3 cases:


```
                AREA Example, CODE
                ENTRY
                BL   func1
stop            B    stop
Func1       1.  LDR r0, =42
            2.  LDR r1, =0x55555555
            3.  LDR r2, =0xFFFFFFFF
                MOV PC, lr
```

```
                AREA Example, CODE
                ENTRY
                BL   func1
stop            B    stop
Func1           MOV r0, #42
                LDR r1, [PC, #4]
                MVN r2, #0
                MOV PC, lr
Lit_pool        DCD 0x55555555
```

Loading constants: LDR → MOV

- Case 1: the instruction `LDR r0, =42` is replaced with `MOV r0, #42` without needing any rotation.


```
            AREA Example, CODE
            ENTRY
            BL   func1
stop        B    stop
Func1       LDR  r0, =42
            LDR  r1, =0x55555555
            LDR  r2, =0xFFFFFFFF
            MOV  PC, lr
```

```
            AREA Example, CODE
            ENTRY
            BL   func1
stop        B    stop
Func1       MOV  r0, #42
            LDR  r1, [PC, #4]
            MVN  r2, #0
            MOV  PC, lr
Lit_pool    DCD  0x55555555
```

Loading constants: LDR → literal pool

- Case 2: the instruction `LDR r1, =0x55555555` is replaced with `LDR r1, [PC, #4]`:
  1. Create a literal pool (Lit_pool) at the end of the code block.
  2. Load the constant by PC-offset addressing (PC + 4).

  Why 4 bytes?
- Allows any 32-bit constants to be loaded into a register.


```
            AREA Example, CODE
            ENTRY
            BL   func1
stop        B    stop
Func1       LDR  r0, =42
            LDR  r1, =0x55555555
            LDR  r2, =0xFFFFFFFF
            MOV  PC, lr
```

→

```
            AREA Example, CODE
            ENTRY
            BL   func1
stop        B    stop
Func1       MOV  r0, #42
            LDR  r1, [PC, #4]    PC – 8 (execute)
            MVN  r2, #0          PC – 4 (decode)
            MOV  PC, lr          PC (fetch)
Lit_pool    DCD  0x55555555      PC + 4
```

Loading constants: LDR → MVN

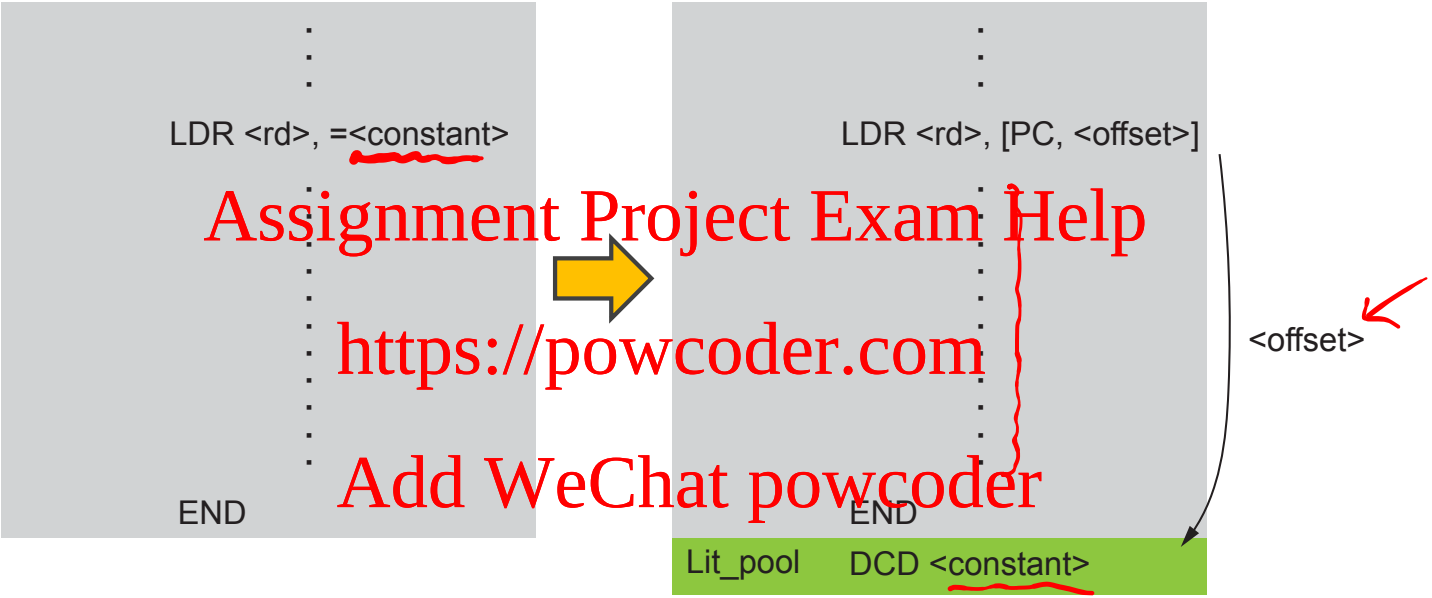
- Case 3: the instruction `LDR r2, =0xFFFFFFFF` is replaced with `MVN r2, #0` without needing any rotation, but requires bit flipping, hence the MVN instruction.


```
            AREA Example, CODE
            ENTRY
            BL   func1
stop        B    stop
Func1       LDR r0, =42
            LDR r1, =0x55555555
            LDR r2, =0xFFFFFFFF
            MOV PC, lr
```

```
            AREA Example, CODE
            ENTRY
            BL   func1
stop        B    stop
Func1       MOV r0, #42
            LDR r1, [PC, #4]
            MVN r2, #0
            MOV PC, lr
Lit_pool    DCD 0x55555555
```

Loading constants: LDR → literal pool

- The literal pool method uses pre-indexed addressing (*PC* + *offset*) to reach the 32-bit constant.



- Limitations:
  - <*offset*> is 12-bit long: max of 4 KB jump in program space.
  - Fails if the END directive is > 4 KB from the `LDR <rd>, =<constant>`.

	31 30 29 28	27	26	25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
Single data transfer	Cond	0	1	1	P	U	B	W	L	Rn	Rd	Offset

12 bits

Loading constants: LDR → literal pool

- Example: literal pool does not work here:

```
                AREA Example, CODE
                ENTRY
                BL    func1
stop            B     stop
Func1           LDR   r0, =42
                LDR   r1, =0x55555555
                LDR   r2, =0xFFFFFFFF
                ADD   r0, r0,r2
                MOV   PC, lr
BigTable        SPACE 4200
                END
```


- `LDR r1, =0x55555555` fails, because END is (~ 4 KB + 8) away, exceeding the limit of pre-indexed addressing.
- What to do?

Loading constants: LTORG

- Already using software (assembler) to re-write code… Take it a step further!
- The **LTORG** directive tells the assembler to build the current literal pool immediately, at where the LTORG appears.

Original code

```
            AREA Example, CODE
            ENTRY
            BL   func1
stop        B    stop
Func1       LDR  r0, =42
            LDR  r1, =0x55555555
            LDR  r2, =0xFFFFFFFF
            ADD  r0, r0,r2
            MOV  PC, lr
            LTORG
BigTable    SPACE 4200
            END
```


Assembler step-1 (build literal pool)

```
            AREA Example, CODE
            ENTRY
            BL   func1
stop        B    stop
Func1       LDR  r0, =42
            LDR  r1, =0x55555555
            LDR  r2, =0xFFFFFFFF
            ADD  r0, r0,r2
            MOV  PC, lr
Lit_pool    DCD 0x55555555
BigTable    SPACE 4200
            END
```

Loading constants: LTORG

- Already using software (assembler) to re-write code… Take it a step further!
- The **LTORG** directive tells the assembler to build the current literal pool immediately, at where the LTORG appears.

Assembler step-2 (re-write)

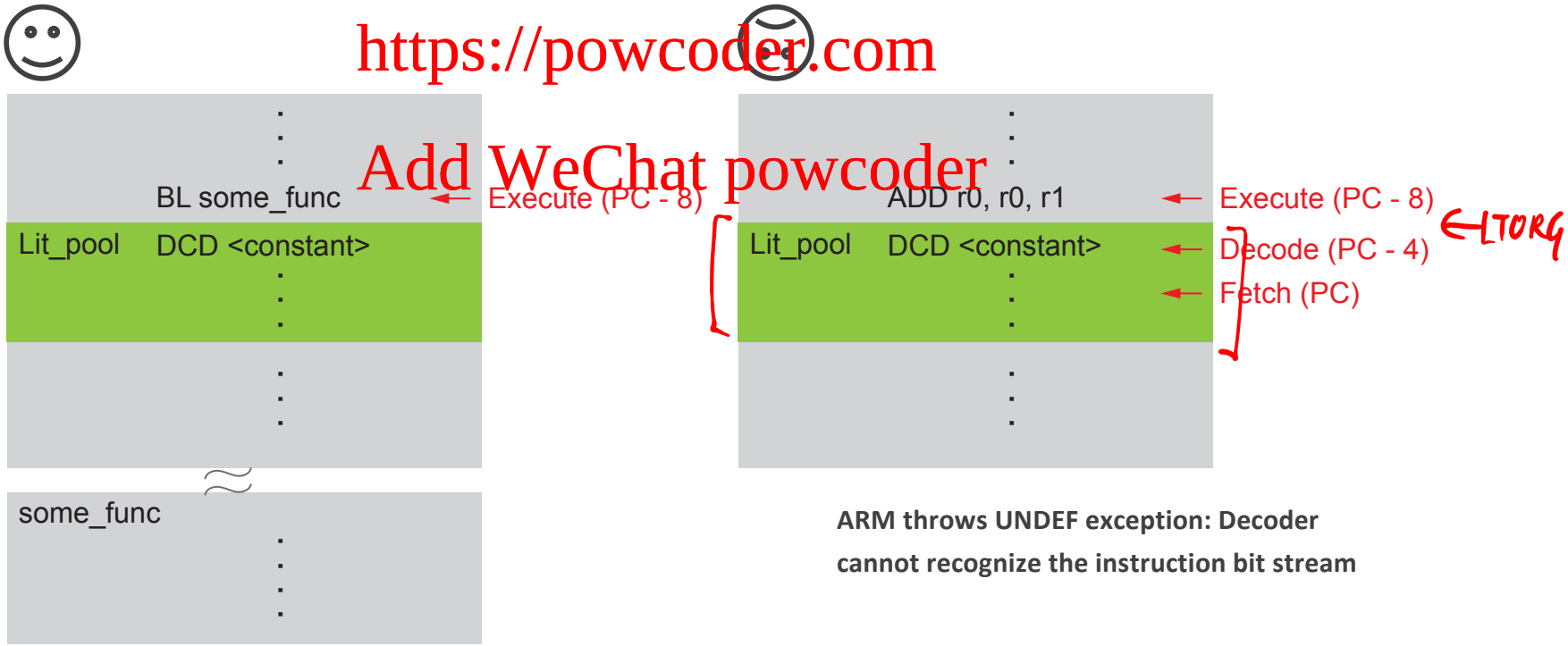
```
                AREA Example, CODE
                ENTRY
                BL   func1
stop            B    stop
Func1           MOV  r0, #42
        Execute LDR  r1, [PC, #8]
        Fetch   MVN  r2, #0
        Decode  ADD  r0, r0,r2     PC
                MOV  PC, lr        PC + 4
Lit_pool        DCD 0x55555555     PC + 8
BigTable        SPACE 4200
                END
```


Assembler step-1 (build literal pool)

```
                AREA Example, CODE
                ENTRY
                BL   func1
stop            B    stop
Func1           LDR  r0, =42
                LDR  r1, =0x55555555
                LDR  r2, =0xFFFFFFFF
                ADD  r0, r0,r2
                MOV  PC, lr
Lit_pool        DCD 0x55555555
BigTable        SPACE 4200
                END
```

Loading constants: LTORG

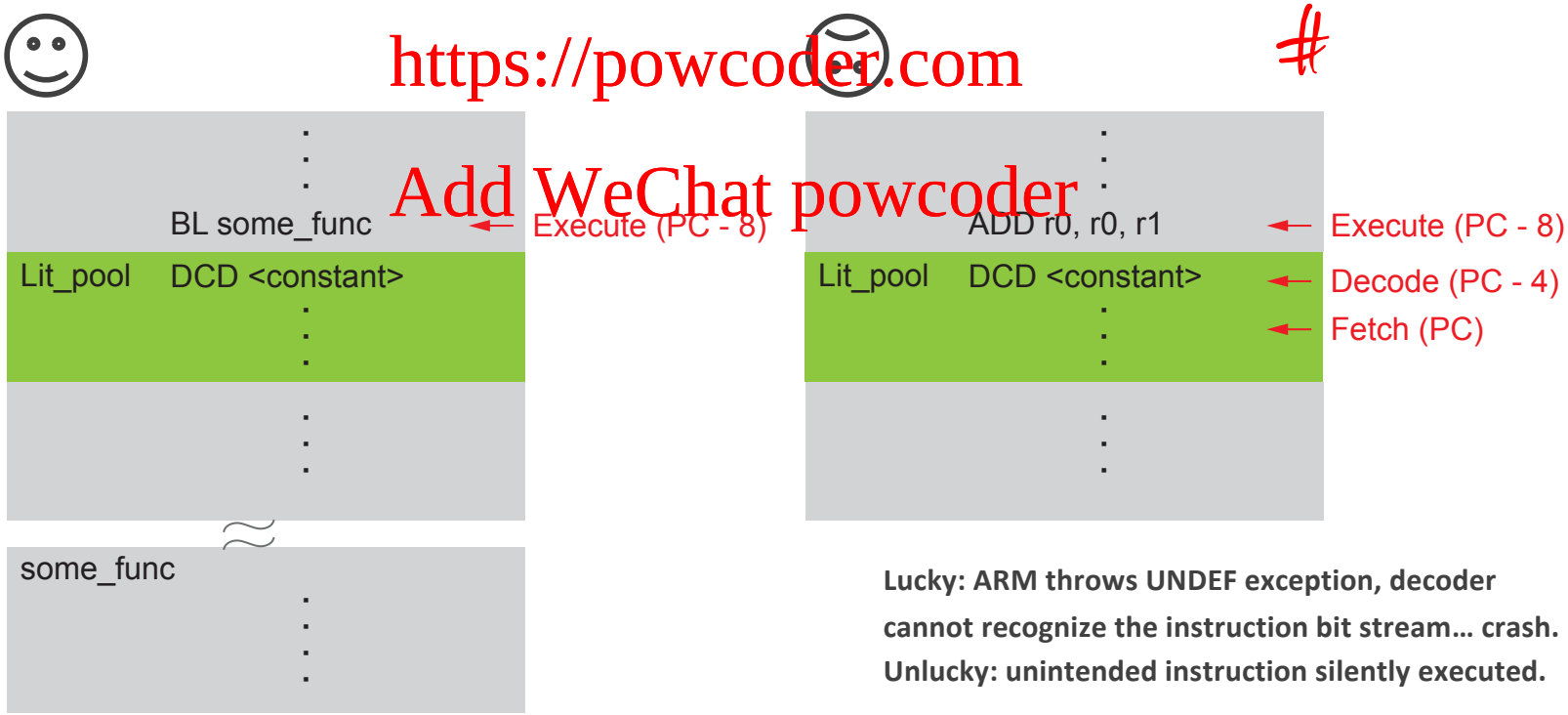
- Be careful where you use LTORG
  - The assembler simply replaces it with a DCD block.
  - Very important that the fetch circuit never enter this block by automatic PC stepping. Otherwise executing the constants as instructions.



ARM throws UNDEF exception: Decoder cannot recognize the instruction bit stream

Loading constants: LTORG

- Best places for LTORG are:
  1. At the end of a subroutine; or ---- MOV PC LR
  2. After unconditional branch instructions ---- B somewhere



Lucky: ARM throws UNDEF exception, decoder cannot recognize the instruction bit stream… crash.
Unlucky: unintended instruction silently executed.

Loading constants: summary

- Use `LDR <rd>, =<constant>` to put **any 32-bit constant** into a register.
- Literal pools are generated at the end of each code section.
- The assembler will (in this order):
  1. If possible, replace with MOV or MVN.
  2. Check if the constant is already in a literal pool. If so, address the existing constant.
  3. Try to place the constant in the next literal pool.
  4. Generate an error if the literal pool will be out of range.
- You can control the literal pool location with `LTORG`.

- LTORG usage:
  - Put LTORG after a subroutine or after an unconditional branch.
  - The LTORG must still be reachable within ± 4KB (PC-offset addressing limit).
    … bottom line: modularizing your code is a good thing.

Loading addresses: ADR

- The starting address of a table / list / set of coefficients is needed to access the table / list / set.
- This address is stored in a register for pre-index / post-index addressing.
- Pseudo-instruction `ADR <rd>, <label>` stores the address of <label> into register <rd>.

```
SRAM_BASE    EQU 0x40000000
             AREA FILTER, CODE
             ENTRY
Main         LDR r0, =#SRAM_BASE
             MOV r3, r0
             ADR r1, Image_data
             ADR r2, Cosines
             BL filter
             .
             .
             ALIGN
Image_data   DCW 0x0001, 0x0002, 0x0003,0x0004
             DCW 0x0005, 0x0006,0x0007, 0x0008
             .
             .
Cosines      DCW 0x3ec5, 0x3537, 0x238e, 0x0c7c
             DCW 0xf384, 0xdc72, 0xcac9, 0xc13b
             .
             .
             END
```

Loading addresses: ADR

- The assembler replaces `ADR <rd>, <label>` with:
  `ADD <rd>, PC, #<offset>` or
  `SUB <rd>, PC, #<offset>`
  - *<offset>* Difference in byte(s) between the PC and the label.
  - *PC* Address of the instruction being fetched (8 bytes ahead of ADR).

- ADR frees us from having to manually track label addresses, letting the assembler update them automatically. Much like `#define` in C.

Loading addresses: ADR

- The assembler computes the PC-relative offset for us.

```
                AREA adrlabel, CODE READONLY
                ENTRY
Start           BL  func
stop            B   stop

func            ADR r0, Start ; SUB r0, PC, #16
                ADR r1, DataArea
                ADD r2, r2, r3
                MOV pc, lr
DataArea        SPACE 8000
                END
```

3
2
1

-16 bytes

Execute (PC-8)

Decode (PC-4)

Fetch (PC)

Loading addresses: ADR

- Q: what #offset would the assembler use when substituting `ADR r1, DataArea`?

```
                AREA adrlabel, CODE READONLY
                ENTRY
Start           BL  func
stop            B   stop

func            ADR r0, Start
                ADR r1, DataArea ⇢ ADD r1, PC, ??
                ADD r2, r2, r3
                MOV pc, lr
DataArea        SPACE 8000
                END
```

Execute (PC-8)

Decode (PC-4)

Fetch (PC)

+4 bytes

`ADD r1, PC, #4`

Loading addresses: ADR

- Pseudo-instruction ADR is converted to real instruction ADD / SUB, with immediate addressing mode. Thus the **offset is limited to 255 words**.
- ADR fails if the label is > 255 words away from the PC. The assembler raises an error.
- How do we work around this limitation?

Loading addresses: ADRL

- Solution: more instruction re-writing tricks ☺
- Assembler replaces `ADRL <rd>, <label>` with **two** ADD / SUB instructions.


```
            AREA adrlabel, CODE READONLY
            ENTRY
Start       BL  func
stop        B   stop

func        ADR r0, Start            ; SUB r0, PC, #16
            ADR r1, DataArea         ; ADD r1, PC, #4
            ADRL r2, DataArea+4300   ⇢ ADD r2, PC, #255
            MOV pc, lr                 ADD r2, r2, #4045
DataArea    SPACE 8000
            END
```

Loading addresses: ADR & ADRL

- Limitation: label used with ADR or ADRL **must be in the same code section**.
  - A code section is denoted by the **AREA** directive.
  - **Sections** are independent, named, indivisible chunks of code or data that are manipulated by the linker, which combines AREAs into a single program.


```
            AREA adrlabel, CODE READONLY
            ENTRY
Start       BL  func
stop        B   stop

func        ADR r0, Start
            ADR r1, DataArea
            ADR r2, DataArea+4300
            MOV pc, lr
DataArea    SPACE 8000
            END
```

A code section

Loading addresses: LDR

- The LDR instruction (previously for loading constants) can also load label addresses.
- The pseudo-instruction has the form: `LDR <rd>, =<label>`
- Unlike ADR and ADRL, **LDR can load addresses outside the current section**.
  - Achieved using literal pools.

Loading addresses: summary

- When loading label address into a register, if the label address is:
  - ≤ 255 words away: use `ADR <rd>, <label>`.
  - > 255 words: use `ADRL <rd>, <label>`.
- Use `LDR <rd>, =<label>` if:
  1. referencing labels outside the current code section, or
  2. you know a literal pool will be created for the current code section

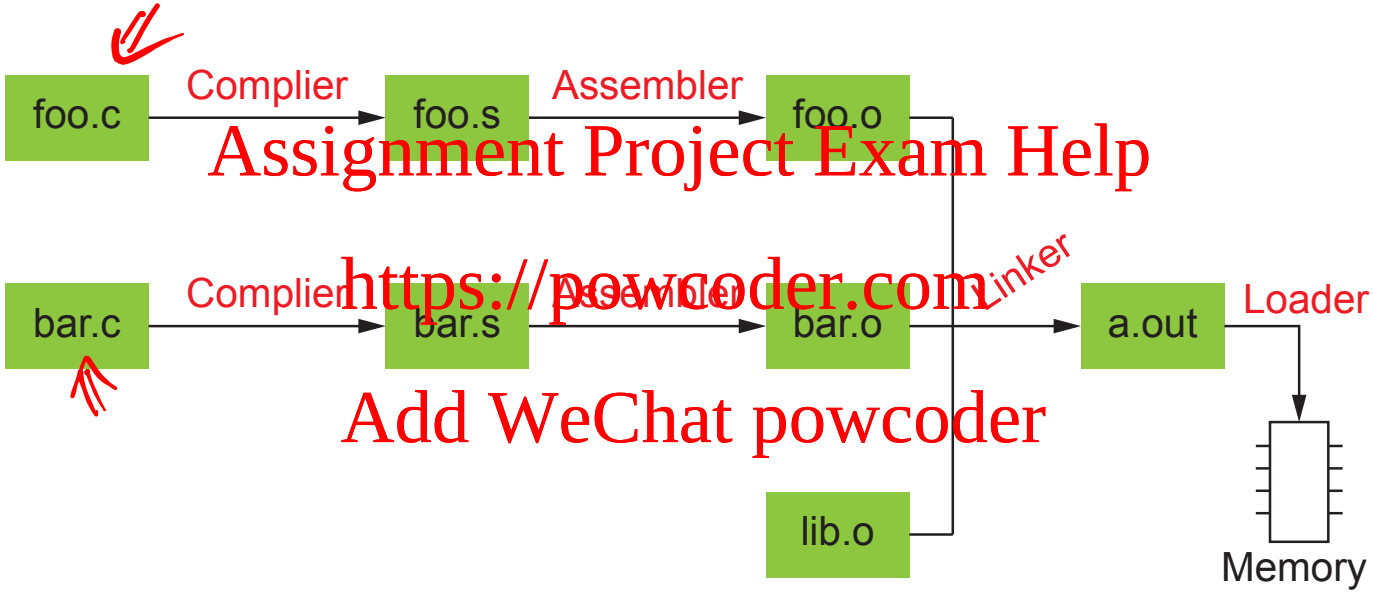
- Q: So complicated, why bother with these guidelines?

Minimizing the amount of code expansion from pseudo-instructions, keeping the program small. Embedded systems have limited memory.

ADR	1 instr
ADRL	2 instr
LDR =<label>	1 instr. + literal pool

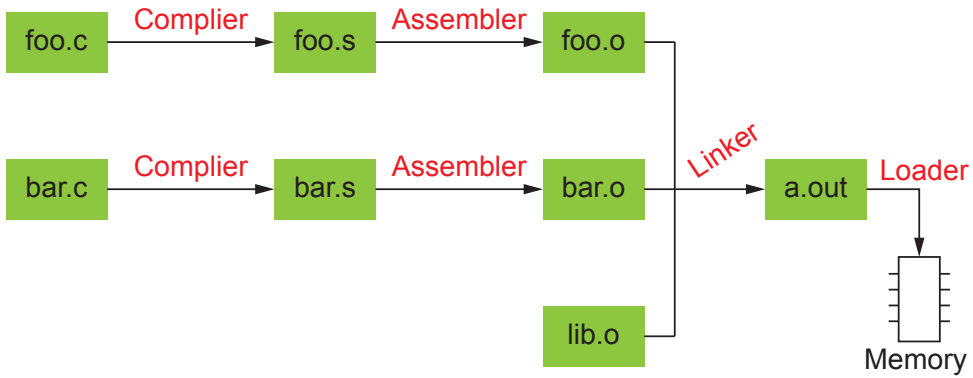
Translating C code to executable

gcc -c ___.c → ___.o



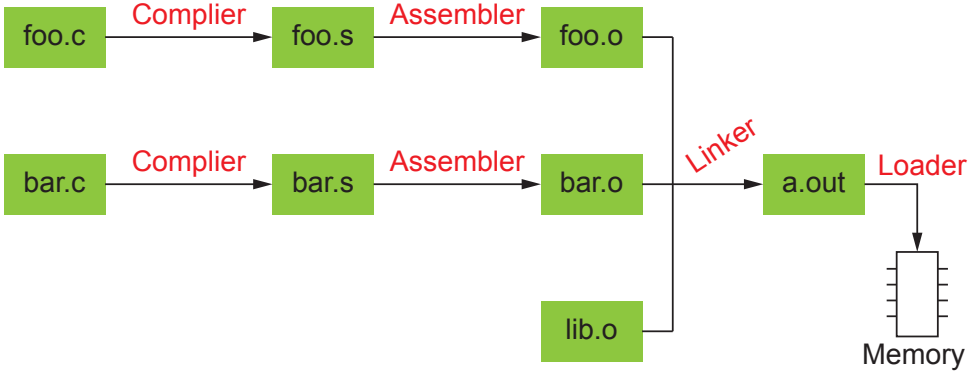
Assembler

- A program that translates symbolic machine instructions into binary representation.
- Main tasks:
  1. Reads and uses directives.
  2. Replaces pseudo-instructions.
  3. Produce machine code
     - » Encodes code and data as bit-blocks from symbolic instruction & declarations
     - » Maps labels into addresses
  4. Creates **object file** (*.o files) - a module.
     - » Contains relocation information, symbol table(s) and (optionally) debugging information.



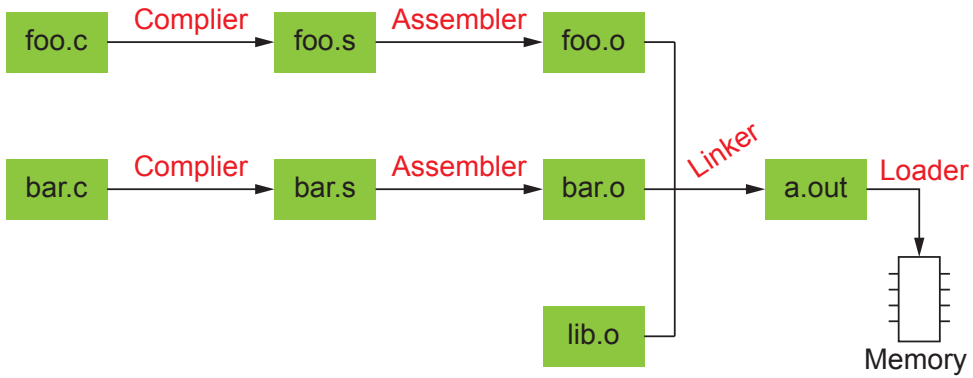
Linker

- A program that combines several object (.o) files into a binary file, executable by the hardware.
- Enable separate compilation of files (modules) – changes to one file do not require recompilation of other files.
- Steps in linking:
  1. Combine text (i.e code) segment from all .o file.
  2. Combine data segment from all .o file, append to the end of the text segment.
  3. Resolve label references, e.g. labels from other code sections.



More directives

- Labels are not visible beyond the current module (*.o file) by default.
- `EXPORT` / `GLOBAL`: declares that a label can be used by the Linker to resolve labels referenced in another module.
  - Makes functions, constants visible to other modules.
- `IMPORT` / `EXTERN`: tells the Assembler that the label is defined elsewhere and will be resolved by the Linker.
  - Allows externally defined functions, constants to be used.



This week

- Constants
- Pseudo-instructions
- Loading constants
- Loading addresses
- Assembler & linker

In Moodle:

- Start working on Lab 5 (Due: end of your 3-hr lab)
- Start doing Week 9 exercise