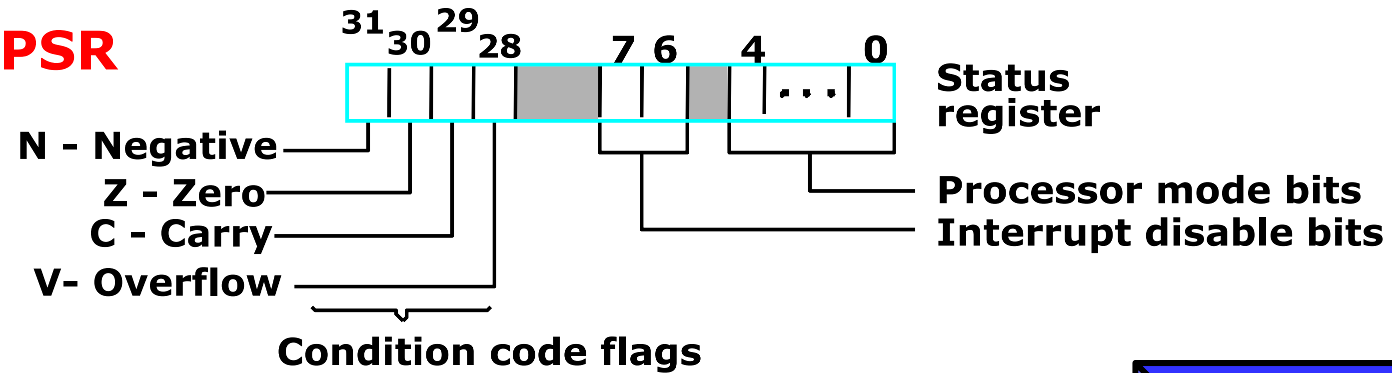
10 ARM Programming 1
CSC 230

Department of Computer Science
University of Victoria

Stallings chapters 12,13 (skip Intel portions)
M&H: chapter 4 translated from ARC
ARM Manual

CPSR



Bit	Name	Purpose
0-4	Mode	I/O (see later)
6-7	Interrupt	Interrupt mask (later)
28	V	Overflow: =1 if arithmetic overflows, else 0
29	C	Carry: =1 if carry-out results from operation, else 0
30	Z	Zero: =1 if result from operation is 0, else 0
31	N	Negative:=1 if result from operation is negative, else 0

Review

N, Z, V and C are set by operations and then used for decision making in branch instructions.

Assembly Language Programming

- ❑ **Assembly language = symbolic form of machine language**
  ➔ Specific to a particular processor type

- ❑ **An assembler is a relatively simple *system* program that translates symbolic assembly language to numeric machine language**

An assembler is much simpler than a compiler because:

- ✓ **assembly language syntax and semantics are much simpler than high-level languages;**
- ✓ **the program can be translated essentially line by line;**
- ✓ **there aren't the issues of context and structure that a high-level language compiler has to deal with.**

Assembly Language Programming

An assembly language program consists of a sequence of statements including:

assembler language instructions:

each corresponds to an executable machine language instruction

assembler directives:

they provide direction to the assembler; e.g. used for creating data areas, determining the placement of code and data in memory etc.

macros:

Some assemblers support macros - a single statement which may be expanded into many

(1) Given: general architectural framework

➔ General syntax and semantics for assembly languages

Initial chapters in books

(2) Given: choice of processor

➔ Processor syntax and semantics for THE GENERAL assembly language for that processor

In ARM Reference books

(3) Given: choice of processor and an implementation of an Assembler

➔ syntax and semantics for *THE* assembly language for *THAT* processor and Assembler

Example: ARM instructions as used in the GNU Assembler or in ARMSim#

Assembly Language Source Code

LABEL:	OPCODE	OPERAND(s)	COMMENT
optional		as required	*optional*

Use tabs between fields to keep code in nice columns

For many assembly languages, a label must start in column 1

opcode: mnemonic for machine language instruction or an assembler directive

operands: symbols, constants, expressions

➔ must follow appropriate addressing mode

```
@  Sample ARM Instructions and code layout (in Lab as well)
@========= Text (Code) =========
   .text          @ Begin the "text" (code) segment
   .global   _start @ Export "_start" symbolic address
_start:

   ldr    r4,=n      @ Load address of var 'n' in r4
   ldr    r1,[r4]    @ Load value of 'n' from address (in R4)
   mov    r0,#0      @ Initialize R0 to 0 (zero)
   mov    r2,#2      @ Initialize R2 to 2
   add    r0,r0,r1   @ r0 := r0 + r1
   subs   r2,r2,#1   @ r2 := r2 - 1, plus condition codes
   add    r2,r0,r1   @ R2 = R0 + R1
   ldr    r4,=sum    @ Load address of var 'sum'
   str    r2,[r4]    @ Save value of R2 at address of 'sum'
   swi    0x11       @ stop executing
@ ========== Data ==========
   .data     @ Begin the "data" segment, for variables
   .align    @ Next item begins at a word (aligned) address
sum:   .word  0      @allocate 1 word and initialize to 0
n:     .word  5      @allocate 1 word and initialize to 5
   .end
```

```
@  Sample ARM Instructions and code layout (in Lab as well)
@========= Text (Code) =========
   .text        @ Begin the "text" (code) segment
   .global   _start @ Export "_start" symbolic address
_start:

   ldr     r4,=n      @ Load address of var 'n' in r4
   ldr     r1,[r4]    @ Load value of 'n' from address (in R4)
   mov     r0,#0      @ Initialize R0 to 0 (zero)
   mov     r2,#2      @ Initialize R2 to 2
   add     r0,r0,r1   @ r0 := r0 + r1
   subs    r2,r2,#1   @ r2 := r2 - 1, plus condition codes
   add     r2,r0,r1   @ R2 = R0 + R1
   ldr     r4,=sum    @ Load address of var 'sum'
   str     r2,[r4]    @ Save value of R2 at address of 'sum'
   swi     0x11       @ stop executing
@ ========== Data ==========
   .data      @ Begin the "data" segment, for variables
   .align     @ Next item begins at a word (aligned) address
sum:   .word  0      @allocate 1 word and initialize to 0
n:     .word  5      @allocate 1 word and initialize to 5
   .end
```

Instruction Types: not all addressing modes are available to all instructions

Data Processing: Arithmetic, Logic

only in registers

➔ register direct addressing mode

Data movement: Load, Store

index addressing modes with autoincrements

Control flow: Branching

conditions based on CPSR bits

Extra in ARM: Conditional execution

for ALL instructions

it avoids some branching and comparing

Most useful Arithmetic Instructions

```
ADD  {cond}  {S}  Rd, Rn,  <Oprnd2>
    ADD   Rd, Rn, Rm              @Rd = Rn + Rm
    ADD   Rd, Rn, #constant       @Rd = Rn + constant
    ADDS  Rd, Rn, Rm              @Rd = Rn + Rm and set CPSR
    ADDS  Rd, Rn, #constant       @ Rd = Rn + constant;
                                  @ and set CPSR
SUB  {cond}  {S}  Rd, Rn,  <Oprnd2>
                                  @ same as ADD
MUL  {cond}  {S}  Rd, Rn,  Rm
    MUL   Rd, Rn, Rm              @ Rd = Rm x Rn
    MULS  Rd, Rn, Rm              @ Rd = Rm x Rs and set CPSR
                                  @ restriction: Rd must be
                                  @ different from Rm
```

Arithmetic Instructions and examples

ADD: Rd = Rn + Operand2

```
ADD  r3,r1,r2        @r3 = r1 + r2
ADD  r3,r1,#10       @r3 = r1 + 10₁₀
ADD  r1,r1,#0x10     @r1 = r1 + 16₁₀
```

ADDS: Rd = Rn + Operand2 and set CPSR

```
ADDS r3,r1,r2     @r3 = r1 + r2
                  @and set condition codes
```

MUL : Rd = Rn x Rm

```
MUL  r3,r1,r2     @r3 = r1 * r2
                  @Rd not equal to Rm
```

Arithmetic Instructions and examples

SUB: Rd = Rn - Operand2

```
SUB  r3,r1,r2          @r3 = r1 - r2
SUB  r3,r1,#12         @r3 = r1 - 12₁₀
SUB  r3,r1,#0x12       @r3 = r1 - 18₁₀
```

SUBS: Rd = Rn - Operand2 and set CPSR

```
SUBS r3,r1,r2      @r3 = r1 - r2
                   @ and set condition codes
```

RSB: Rd = Operand2 – Rn

```
RSB  r3,r1,r2      @r3 = r2 - r1
                   @useful for compilers
RSB  r3,r1,#0      @r3 = - r1
```

Arithmetic Instructions (Table 1 and Table 2)

Template: Opcode {Cond} {S} Rd, Rn, Rm

ADD	Rd = Rn + Rm
ADDS	Rd = Rn + Rm and set CPSR
ADC	Rd = Rn + Rm + Carry
SUB	Rd = Rn – Rm
SUBS	Rd = Rn - Rm and set CPSR
SBC	Rd = Rn – Rm + Carry – 1
RSB	Rd = Rm - Rn
MUL	Rd = Rm x Rs
MULS	Rd = Rm x Rs and set CPSR
MLA	Rd = Rm x Rs + Rn

Logic Operators

AND

A	B	$C = A \wedge B$
0	0	0
0	1	0
1	0	0
1	1	1

OR

A	B	$C = A + B = A \vee B$
0	0	0
0	1	1
1	0	1
1	1	1

NOT

A	$x = \bar{A} = A'$
0	1
1	0

XOR

A	B	$x = A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

Logic Instructions and examples

AND **Rd = Rn ∧ Operand2**

```
AND  r3,r3,r1      @r3 = r3 ∧ r1
AND  r3,r3,#0x10   @r3 = r3 ∧ 10₁₆
```

0200 0008

0000 0000

self learning!

OR **Rd = Rn ∨ Operand2**

```
ORR  r3,r3,r1      @r3 = r3 ∨ r1
ORR  r3,r3,#0x10   @r3 = r3 ∨ 10₁₆
```

0234 5618

0204 0618

Example:

r3 = 0x 0204 0608   r1 = 0x 0230 5018

Logic Instructions (Table 3)

Template: Opcode {Cond} Rd, Rn, Operand2

AND	**Rd = Rn $\wedge$ Operand2**
ORR	**Rd = Rn $\vee$ Operand2**
EOR	**Rd = Rn $\oplus$ Operand2**
BIC	**Rd = Rn $\wedge \neg$ Operand2**
CMP	**test (Rn – Operand2) to set CPSR**
CMN	**test (Rn + Operand2) to set CPSR**
TST	**test (Rn $\wedge$ Operand2) to set CPSR**
TEQ	**test (Rn $\oplus$ Operand2) to set CPSR**
MVN	**Rd = $\neg$ Operand2 @move complement (negated)**

Moving data (Tables 4 and 7)

Template: Opcode {Cond} Rd, Operand2

MOV	**Rd = Operand2**	**@register or constant**
MVN	**Rd = ¬ Operand2**	**@move complement (negated)**
LDR	**Rd = EA**	**@load word from memory**
LDRB	**Rd = EA**	**@load byte from memory**
STR	**EA = Rd**	**@store word in memory**
STRB	**EA = Rd**	**@store byte in memory**

- **Here the correct choice of addressing modes is crucial**
- **There are instructions to Load/Store multiple items in Table B.5 (later)**

Branch Instructions (Table 12)

Template: Opcode Label

BEQ	**Equal (zero)**	**Z=1**	
BNE	**Not equal (zero)**	**Z=0**	
BGE	**Signed greater or equal**		*use for data*
BLT	**Signed less**		
BGT	**Signed greater**		
BLE	**Signed less or equal**		
BHI	**Unsigned higher**		*use for addresses*
BLS	**Unsigned lower or same**		
BAL	**Always**		

Full List of ARM Condition Mnemonics

EQ	Equal	Z=1
NE	Not equal	Z=0
CS/HS	Carry Set/Unsigned higher or same	C=1
CC/LO	Carry Clear/Unsigned lower	C=0
MI	Minus/Negative	N=1
PL	Plus/Positive or Zero	N=0
VS	Overflow	V=1
VC	No overflow	V=0
HI	Unsigned higher	C=1 & Z=0
LS	Unsigned lower or same	C=0 & Z=1
GE	Signed greater or equal	N=V
LT	Signed less than	N≠V
GT	Signed greater	Z=0 & N=V
LE	Signed less or equal	Z=1 & N≠V
AL	Always	true

From Local ARM Reference Card.
(Available on both the midterm and the final)

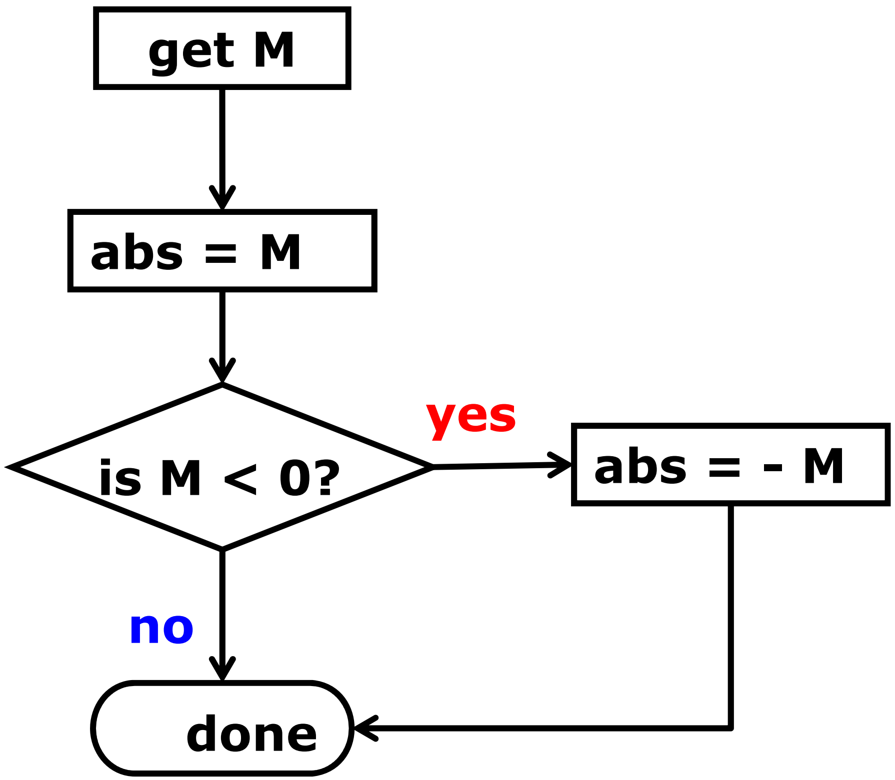
Operation	Assembler	Action
Move	MOV{S} Rd, <Oprnd2>	Rd := Oprnd2 {CPSR}
	MVN{S} Rd, <Oprnd2>	Rd := NOT Oprnd2 {CPSR}
Arithmetic	ADD{S} Rd, Rn, <Oprnd2>	Rd := Rn + Oprnd2 {CPSR}
	ADC{S} Rd, Rn, <Oprnd2>	Rd := Rn + Oprnd2 + Carry {CPSR}
	SUB{S} Rd, Rn, <Oprnd2>	Rd := Rn - Oprnd2 {CPSR}
	SBC{S} Rd, Rn, <Oprnd2>	Rd := Rn + Oprnd2 + Carry {CPSR}
	RSB{S} Rd, Rn, <Oprnd2>	Rd := Oprnd2 - Rn {CPSR}
	RSC{S} Rd, Rn, <Oprnd2>	Rd := Oprnd2 - Rn - NOTCarry {CPSR}
	MUL{S} Rd, Rm, Rs	Rd := Rm * Rs {CPSR}
	MLA{S} Rd, Rm, Rs, Rn	Rd := Rm * Rs + Rn {CPSR}
	CLZ Rd, Rm	Rd := # leading zero in Rm
Logical	AND{S} Rd, Rn, <Oprnd2>	Rd := Rn AND Oprnd2 {CPSR}
	EOR{S} Rd, Rn, <Oprnd2>	Rd := Rn EXOR Oprnd2 {CPSR}
	ORR{S} Rd, Rn, <Oprnd2>	Rd := Rn OR Oprnd2 {CPSR}
	TST Rn, <Oprnd2>	Update CPSR on Rn AND Oprnd2
	TEQ Rn, <Oprnd2>	Update CPSR on Rn EOR Oprnd2
	BIC{S} Rd, Rn, <Oprnd2>	Rd := Rn AND NOT Oprnd2 {CPSR}
	NOP	R0 := R0
Compare	CMP Rd, <Oprnd2>	Update CPSR on Rn - Oprnd2
Branch	B{cond} label	R15 := label
	BL{cond} label	R14 := R15-4; R15 := label
Swap	SWP Rd, Rm	temp := Rn; Rn := Rm; Rd := temp
Load	LDR Rd, <a_mode2>	Rd := address
	LDM <a_mode4L> Rd{!}, <reglist>	Load list of registers from [Rd]
Store	STR Rd, <a_mode2>	[address]:= Rd
	STM <a_mode4S> Rd{!}, <reglist>	Store list of registers to [Rd]
SWI	SWI <immed_24>	Software Interrupt

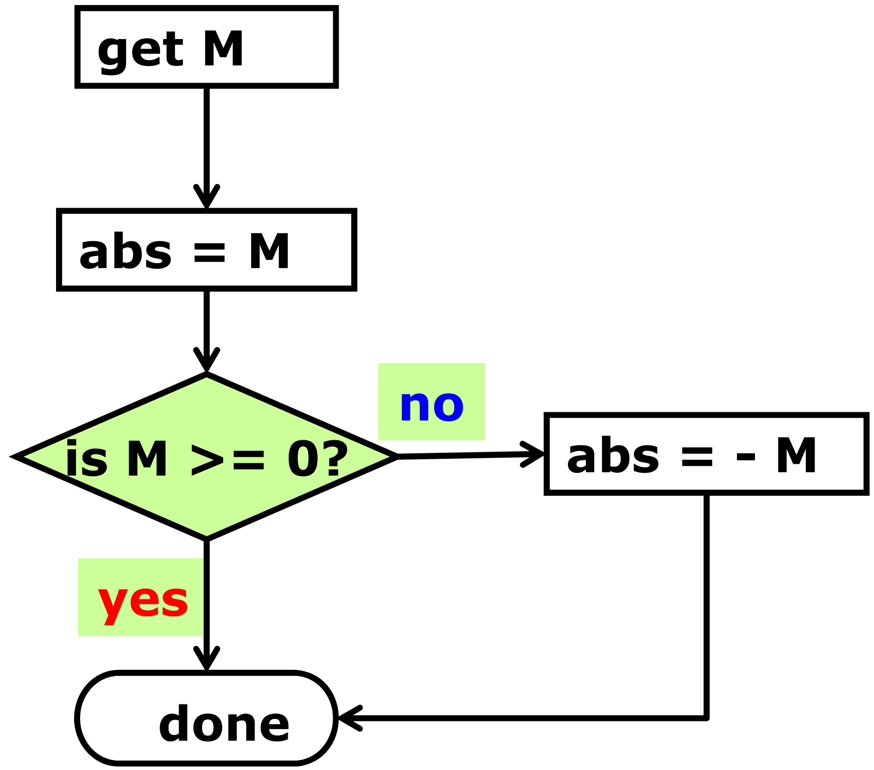
Addressing Mode 2 - Data Transfer		
Pre-indexed	Immediate offset	[Rn, #+/-<immed_12>]{!}
	Zero offset	[Rn]
	Register offset	[Rn, +/-Rm]{!}
	Scaled register offset	[Rn, +/-Rm, LSL #<immed_5>]{!}
		[Rn, +/-Rm, LSR#<immed_5>]{!}
		[Rn, +/-Rm, ASR #<immed_5>]{!}
		[Rn, +/-Rm, ROR #<immed_5>]{!}
		[Rn, +/-Rm, RRX]{!}
Post-indexed	Immediate offset	[Rn], #+/-<immed_12>
	Register offset	[Rn], +/-Rm
	Zero offset	[Rn]
	Scaled register offset	[Rn], +/-Rm, LSL #<immed_5>
		[Rn], +/-Rm, LSR #<immed_5>
		[Rn], +/-Rm, ASR #<immed_5>
		[Rn], +/-Rm, ROR #<immed_5>
		[Rn], +/-Rm, RRX

Key to tables	
{cond}	See Condition Field
<Oprnd2>	See Operand 2
{S}	Updates CPSR if present
<immed>	Constant
<a_mode2>	See Addressing Mode 2
<a_mode4>	See Addressing Mode 4
<reglist>	List of registers with commas
{!}	Updates base register if present

Condition Field	
EQ	Equal
NE	Not equal
CS	Carry Set
CC	Carry clear
MI	Negative
PL	Positive or zero
VS	Overflow
VC	No overflow
HI	Unsigned higher
LS	Unsigned lower or same
GE	Signed greater or equal
LT	Signed less than
GT	Signed greater than
LE	Signed less than or equal
AL	Always

```
@*** ABSOLUTE VALUE ***

@ Given the number, positive or negative
@ integer, compute ABS(M)

@  Pseudo-code:
@     abs = M;
@     IF M < 0 THEN
@        abs = - M;

@ ========= Text (Code) =========

   .text                      @ Begin code segment
   .global _start       @ For linker
_start:
```





abs = M;
IF M < 0 THEN
abs = - M
return (abs)

IF M >= 0
THEN go to DONEABS
ELSE
abs = - M
DONEABS: ....

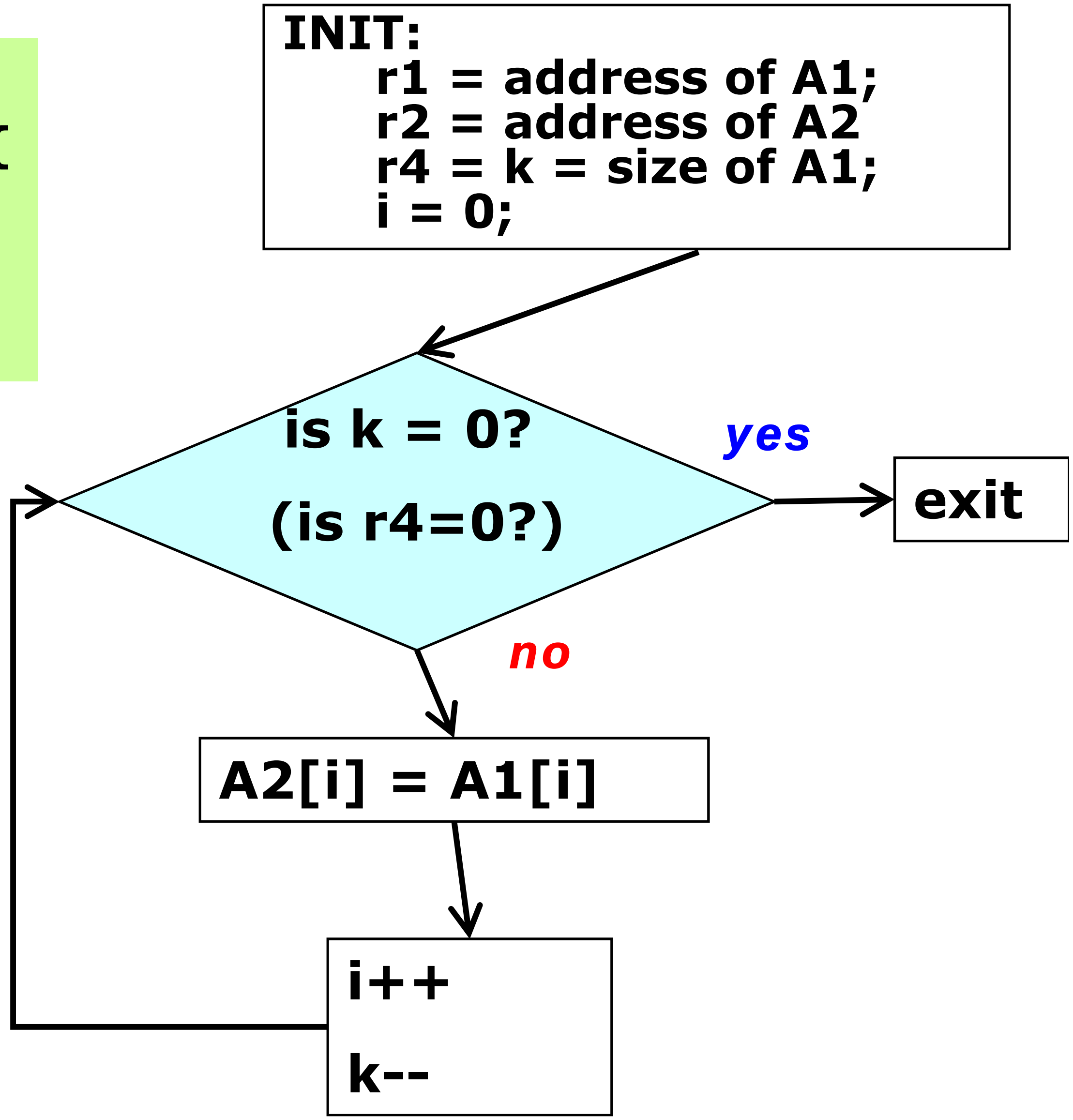
```
@*** ABSOLUTE VALUE ***
@ Given a positive or negative integer M, compute ABS(M)
@  Pseudo-code:
@      abs = M;
@      IF M < 0 THEN
@         abs = -M;
@ ========= Text (Code) =========
    .text           @ Begin the "text" (code) segment
    .global   _start @ Export symbolic address for linker
_start:
       ldr    r0,=M        @r0 = address of M
       ldr    r0,[r0]      @r0 = value of M
       cmp    r0,#0        @ M < 0?
       bpl    doneabs
       mvn    r0,r0        @do 1's complement
       add    r0,r0,#1     @now 2's complement
@      rsb    r0,r0,#0     @2’s complement in 1 instruction
doneabs:  swi    0x11
@ ========== Data ==========
    .data      @ Begin the "data" segment, for variables
    .align     @ Next item is word aligned address
M:  .word      -7
    .end
```

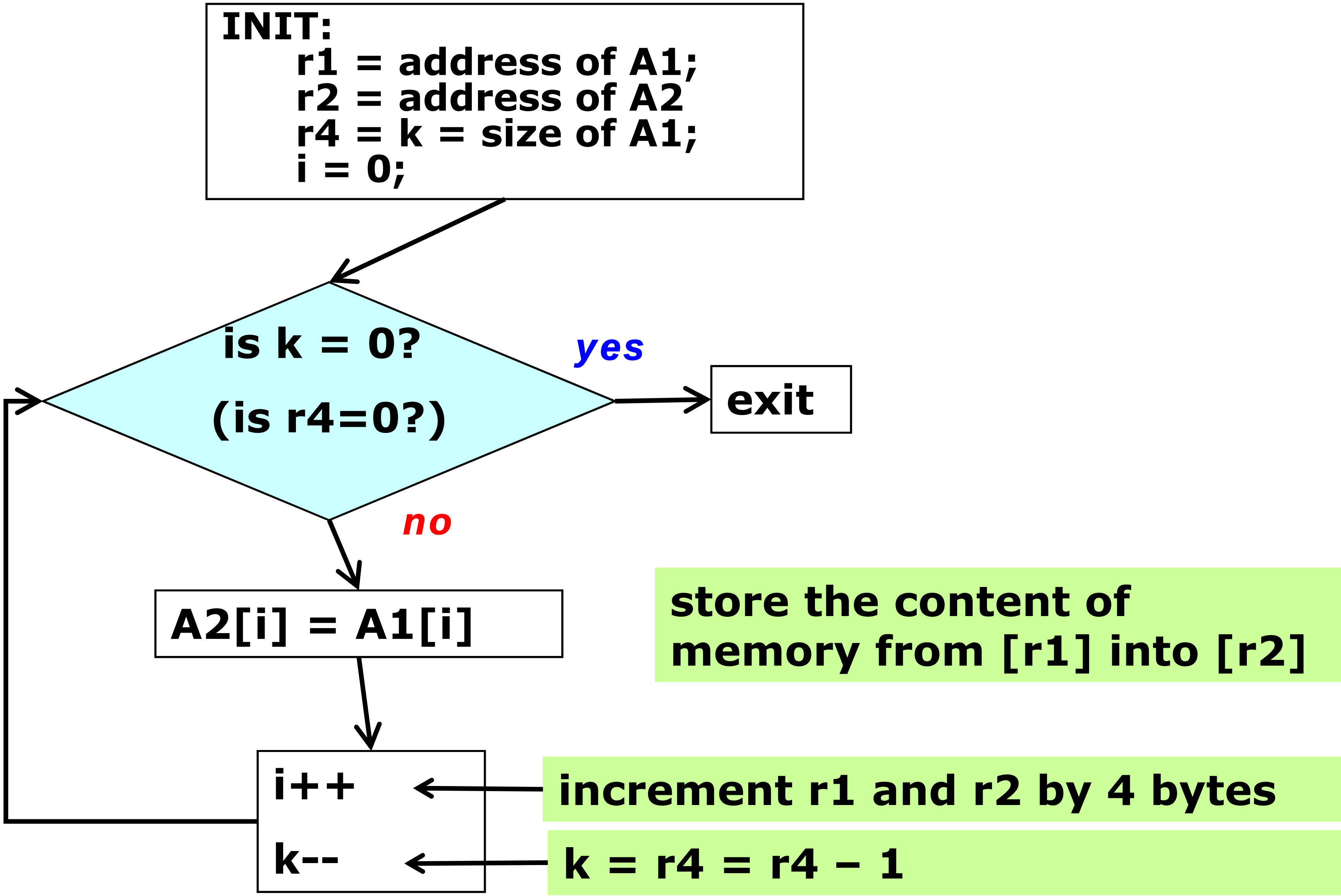
```
_start:
    ldr    r0,=M          @r0 = address of M
    ldr    r0,[r0]        @r0 = value of M

    cmp    r0,#0          @M >= 0?
    bpl    doneabs
    mvn    r0,r0          @do 1's complement
    add    r0,r0,#1       @now 2's complement
@   rsb    r0,r0,#0       @ 2’s complement in 1 instruction
doneabs:
    swi       0x11

@ ========== Data ==========
    .data     @ Begin the "data" segment, for variables
    .align    @ Ensure next item is word aligned
M:  .word -7
    .end
```

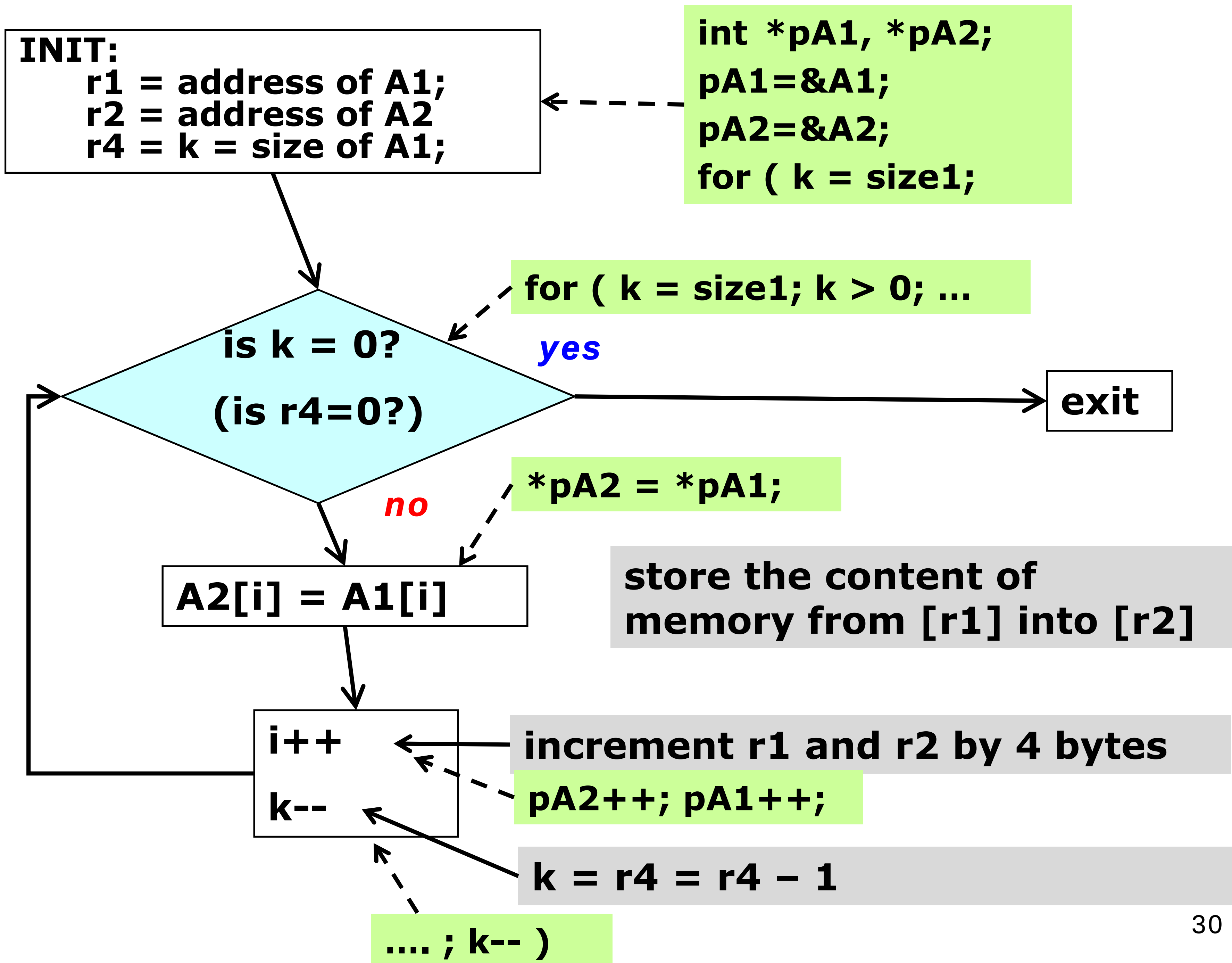
```
@ Arrays examples:arrays and indexes Part 1
@ Get array size S1 for 1st array A1
@ Copy array A1 into array A2, and copy S1 to S2 (size of A2)
@ r1        pointer to array A1
@ r2        pointer to array A2
@ r3        content of element of array
@ r4        size of array 1
@ r5        size of array 2
            .text
            .global   _start
_start:
            ldr       r1,=A1              @r1 := address of A1
            ldr       r2,=A2              @r2 := address of A2
            ldr       r4,=S1              @r4 := address of S1
            ldr       r5,=S2              @r5 := address of S2
            ldr       r4,[r4]             @r4 := size of A1
            str       r4,[r5]             @set size for A2
mainloop:
            cmp       r4,#0               @end of data?
            beq       finish1
            ldr       r3,[r1]             @r3 := element of A1
            str       r3,[r2]             @A2 := copy of element of A1
            add       r1,r1,#4            @move pointer to A1
            add       r2,r2,#4            @move pointer to A2
            sub       r4,r4,#1            @subtract counter
            bal       mainloop
finish1:    swi       0x11
            .data               @ Begin the "data" segment, for variables
            .align              @ Ensure that the next item has a word aligned address
S1:         .word     10        @size of A1
S2:         .word     0         @size of A2
A1:         .word     1,9,7,3,2,5,8,4,0,6         @ array 1
A2:         .skip     40                          @ array 2
            .end
```

```
i = 0
for ( k = size1; k > 0; k-- ) {
     A2[i] = A1[i]
     i++
}
```

```
int *pA1, *pA2;
pA1=&A1;
pA2=&A2;
for ( k = size1; k > 0; k-- ) {
     *pA2 = *pA1
     pA2++;
     pA1++;
}
```




INIT:
r1 = address of A1;
r2 = address of A2
r4 = k = size of A1;
i = 0;
is k = 0?
(is r4=0?)
yes
exit
no
A2[i] = A1[i]
store the content of memory from [r1] into [r2]
i++
k--
increment r1 and r2 by 4 bytes
k = r4 = r4 – 1


INIT:
r1 = address of A1;
r2 = address of A2
r4 = k = size of A1;
int *pA1, *pA2;
pA1=&A1;
pA2=&A2;
for (k = size1;
for (k = size1; k > 0; …
is k = 0?
(is r4=0?)
yes
exit
no
*pA2 = *pA1;
A2[i] = A1[i]
store the content of memory from [r1] into [r2]
i++
k--
increment r1 and r2 by 4 bytes
pA2++; pA1++;
k = r4 = r4 – 1
…. ; k--)

```
@ Arrays examples:arrays and indexes Part 1

@ Get array size S1 for 1st array A1
@Copy array A1 into array A2,copy S1 to S2 (size of A2)

@ r1  pointer to array A1
@ r2  pointer to array A2
@ r3  content of element of array
@ r4  size of array 1
@ r5  size of array 2

      .text
      .global     _start

_start:
```

Document register usage for yourself!

```
        ldr   r1,=A1             @r1 := address of A1
        ldr   r2,=A2             @r2 := address of A2
        ldr   r4,=S1             @r4 := address of S1
        ldr   r4,[r4]            @r4 := size of A1
        ldr   r5,=S2             @r5 := address of S2
        str   r4,[r5]            @set size for A2
mainloop:
        cmp   r4,#0              @end of data?
        beq   finish1
        ldr   r3,[r1]      @r3 := element of A1
        str   r3,[r2]      @A2 := copy of element of A1
        add   r1,r1,#4     @move pointer to A1
        add   r2,r2,#4     @move pointer to A2
        sub   r4,r4,#1     @subtract counter
        bal   mainloop
finish1:      swi   0x11
        .data
S1:     .word  10     @size of A1
S2:     .word  0      @size of A2
A1:     .word  1,9,7,3,2,5,8,4,0,6        @ array A1
A2:     .skip  40                         @ array A2
```

```
        ldr   r1,=A1               @r1 := address of A1
        ldr   r2,=A2               @r2 := address of A2
        ldr   r4,=S1               @r4 := address of S1
        ldr   r5,=S2               @r5 := address of S2
        ldr   r4,[r4]              @r4 := size of A1
        str   r4,[r5]              @set size for A2
mainloop:
        cmp   r4,#0                @end of data?
        beq   finish1
        ldr   r3,[r1],#4     @r3 := element of A1++
                             @ and increase pointer
        str   r3,[r2],#4     @A2 := copy of element of A1++
                             @ and increase pointer
        sub   r4,r4,#1       @subtract counter
        bal   mainloop
finish1:      swi   0x11
        .data
S1:     .word  10     @size of A1
S2:     .word  0      @size of A2
A1:     .word  1,9,7,3,2,5,8,4,0,6         @ array A1
A2:     .skip  40                          @ array A2
```

Useful addressing mode!

```
ldr    r3,[r1]       @r3 := element of A1
str    r3,[r2]       @A2 := copy of element of A1
add    r1,r1,#4      @move pointer to A1
add    r2,r2,#4      @move pointer to A2
```

```
ldr    r3,[r1],#4    @r3 <- element of A1 and then
                     @ increase pointer
str    r3,[r2],#4    @A2 <- copy of element of A1 and then
                     @ increase pointer
```

```
        ldr   r1,=A1              @r1 := address of A1
        ldr   r2,=A2              @r2 := address of A2
        ldr   r4,=S1              @r4 := address of S1
        ldr   r5,=S2              @r5 := address of S2
        ldr   r4,[r4]             @r4 := size of A1
        str   r4,[r5]             @set size for A2
mainloop:
        ldr   r3,[r1]       @r3 := element of A1
        str   r3,[r2]       @A2 := copy of element of A1
        add   r1,r1,#4      @move pointer to A1
        add   r2,r2,#4      @move pointer to A2
        subs  r4,r4,#1      @subtract counter
        bne   mainloop      @ end of data?
finish1:      swi   0x11
        .data
S1:     .word 10      @size of A1
S2:     .word 0       @size of A2
A1:     .word 1,9,7,3,2,5,8,4,0,6       @ array A1
A2:     .skip 40                        @ array A2
```

Better loop version

Another possibility

```
_start:
   ldr  r1,=A1    @r1 := address of A1
   ldr  r2,=A2    @r2 := address of A2
   ldr  r4,=S1    @r4 := address of S1
   ldr  r5,=S2    @r5 := address of S2
   ldr  r4,[r4]   @r4 := size of A1
   str  r4,[r5]   @set size for A2
1  mov  r6,#0     @r6 := offset within arrays for indexing
mainloop:
2  cmp  r4,#0          @end of data?
3  beq  finish1
4  ldr  r3,[r1,r6]     @ r3 := element of A1
5  str  r3,[r2,r6]     @ A2 := copy of element of A1
6  add  r6,r6,#4       @ increase offset for indexing
7  sub  r4,r4,#1       @ subtract counter
8  bal  mainloop
```

Advantage: initial pointers to arrays never change!

Assembler Directives

More information is needed besides instructions themselves
➔***Assembler directives or pseudo instructions***

- how to interpret names
- where to place instructions in memory
- where to place data in memory
- linker and loader directives

Usually local to a platform and a particular assembler/compiler

Gnu Assembler Directives

.text where to place the code

.data where to place the data

.end End of *assembler* input
This causes the *assembler to stop* reading the file
END is different from any form of STOP

_start: Label used as program entry point (for linker)

.global _start Export "_start" symbolic address for linker

.align Next allocation in memory must be aligned on word boundary

.word .byte .space .ascii .asciz .skip <size>
allocation of memory space for variables

➔ ***Local ARM Manual for this course***

➔ ***GNU Assembler manual, get ARM section***

Assembler Directives: EQU

- **Use of symbolic labels to represent constant values**
- **It makes programs easier to read and more maintainable**

- *EQU assigns a value to a label.*
- *It does not allocate memory space.*
- *The assembler adds the label to its symbol table.*

```
.equ      MAXSIZE,100        @in ARM
```

➔ MAXSIZE is “equated” to 100
➔ Every time the identifier “MAXSIZE” is found in the program, the assembler substitutes the value “100”.
The label cannot be redefined elsewhere.

Expressions can also be used. For example,

```
.equ      MAX2, 15
.equ      MAXSIZE, MAX2 *2
```

```
@This program sums the first N integers up to a
@ max given value for N
      .text
      .global _start
      .equ  NMAX,20
_start:
@ Register usage:
@ r1  <->   sum of integers
@ r2  <->   max number to decrement from
      MOV   r1,#0       @r1 = sum := 0
      MOV   r2,#NMAX    @r2 := max value
loopadd:
      ADD   r1,r1,r2    @add integer to sum
      SUBS  r2,r2,#1    @decrement integer
      BNE   loopadd
exit: LDR   r2,=sum           @store total
      STR   r1,[r2]
      swi   0x11
      .data
      .align            @ align on a word boundary
sum:  .word 0
      .end
```

NOTES (ABSOLUTELY CRUCIAL)

- Labels are ***NOT*** variables
- Labels are really names for constants
  - Some labels are assigned values by the Assembler
  - For others, the value is not known until loading time
- In some Assembly languages, labels can be redefined or defined multiple times; in others, not.
- In some Assembly languages, labels are **GLOBAL** to an assembly language program and **not LOCAL** to a function or subroutine within the program.
- If a label is assigned a value by an EQU directive, the value can be any number. How the label is used later determines its interpretation.
- For other labels (not defined by EQU) the value assigned to a label is the address of a location in memory and determined by its position in the program.

Reminders from C and pointers

Review

```
int   x, y;     /* integer x */
int   *px;      /*pointer to an integer */

px = &x;     /* "&" is a unary operator which obtains
                the address of a data object */

y = *px;     /* "*" is the operator to dereference from
                a pointer (gets to the data value).
                That is, *px is the object that the
                pointer px points to */
y = x;       /* same as above */
```

in ARM:

LDR	**R1,=X**	**same as**	`px = &x;`
LDR	**R2,[R1]**	**same as**	`y = *px;`

New in ARM

Operations on pointers: Reminders from C and pointers

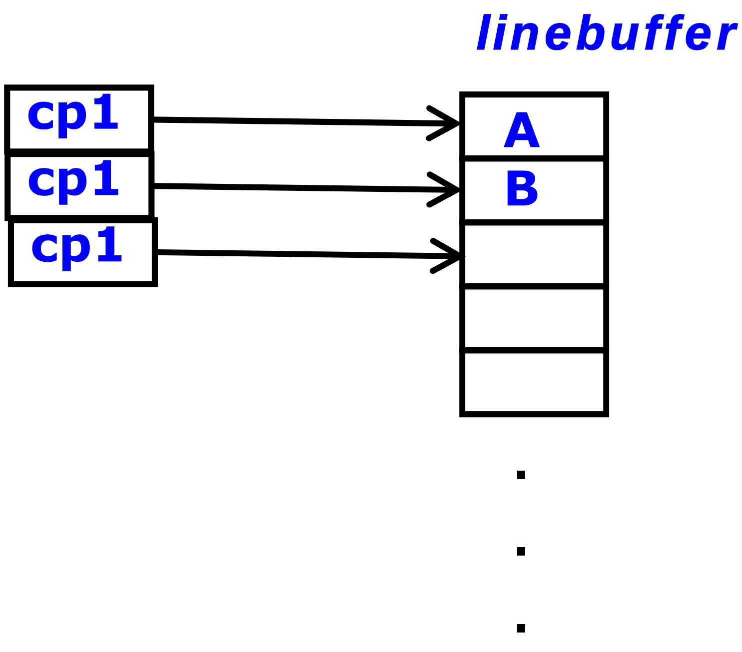
assignment, comparison, add/subtract a constant

Review

```
char  *cp1, linebuffer [100];
.................
cp1 = linebuffer;     /*same as cp1 = &linebuffer[0] */
                      /* cp1 points to first element */
*cp1 = 'A';           /* means linebuffer[0] ='A' */
cp1++;                /*advance to 2nd element*/
*cp1 = 'B';           /* means linebuffer[1] ='B' */
cp1++;                /*advance to 3rd element*/
*cp1 = 'C';           /* means linebuffer[2] ='C' */
...................
```

```
LDR   r1,=linebuffer
MOV   r2,#'A
STRB  r2,[r1]      @linebuffer[0] = A
ADD   r1,r1,#1     @go to second element
MOV   r2,#'B
STRB  r2,[r1],#1   @linebuffer[1]=B and autoincrement..
```

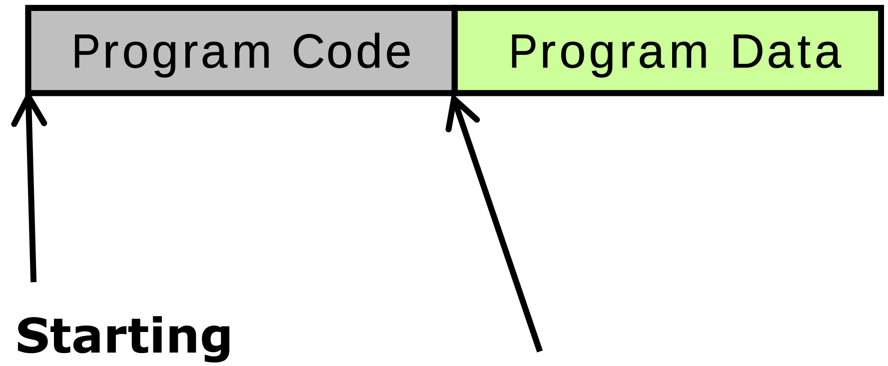
```
char  *cp1, linebuffer [100];
.................
cp1 = linebuffer;     /* same as cp1 = &linebuffer[0] */
                      /* cp1 points to first element */
*cp1++ = 'A';            /* means linebuffer[0] =‘A’
                         and advance to 2nd element*/
*cp1++ = 'B';            /* means linebuffer[1] =‘B’ */
                         and advance to 3rd element*/
....................
```



Using the .data directive

```
_start:
.....
        .data
NUM1: .word  1
```

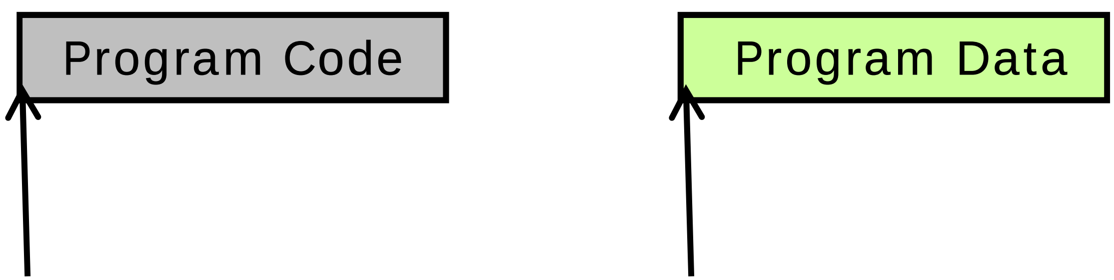
Contiguous Code and Data



Starting address of program

Start + XX bytes

Separate Code and Data



Starting address of program

Starting address of data

Questions / feedback

`LDR R1, [R2]` assuming that R2 already contains an address, how does the processor know that the new content of R1 is now some value? What if it were another address pointing at other data?

Why are registers faster and more expensive than RAM? What is the difference in implementation?

How does the decoder know what the opcode is?

How does a compiler/assembler translate to binary?

How do you test whether a number in a register is odd or even in 1 instruction? (Hint: use a logic instruction)